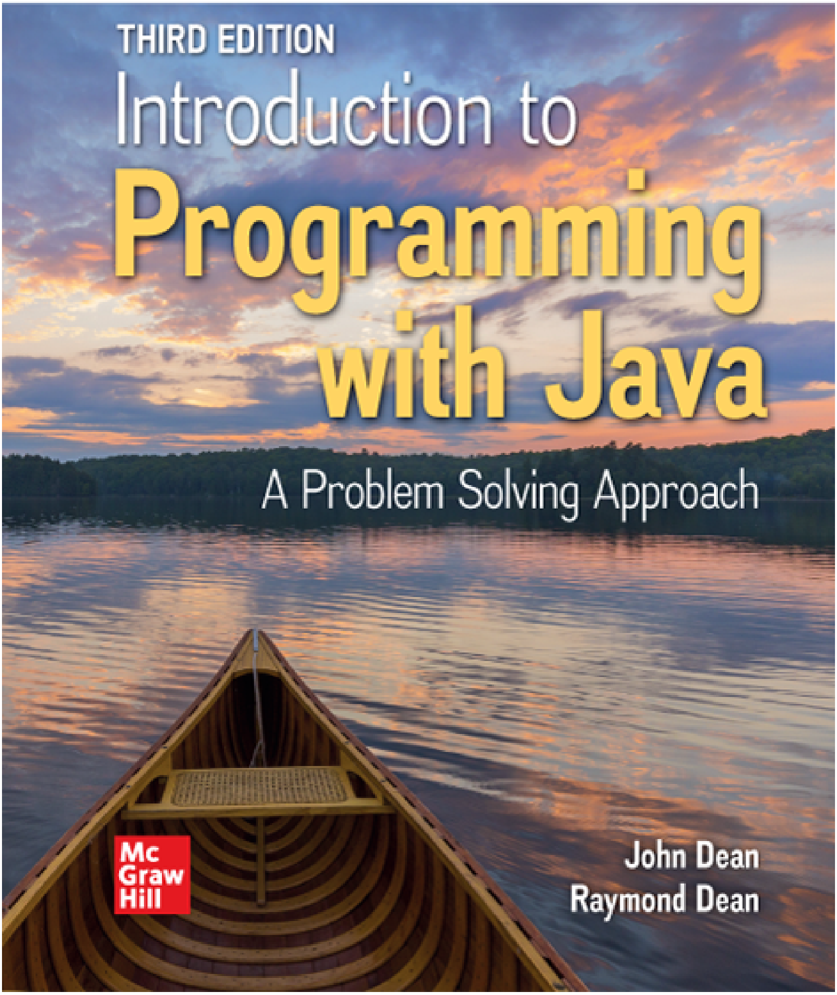
This International Student Edition is for use outside of the U.S.



THIRD EDITION

Introduction to Programming with Java

A Problem Solving Approach

John Dean
Raymond Dean

Introduction to Programming WITH JAVA

A Problem Solving Approach

John Dean
Park University

Raymond Dean
University of Kansas

Mc Graw Hill

INTRODUCTION TO PROGRAMMING WITH JAVA

Published by McGraw Hill LLC, 1325 Avenue of the Americas, New York, NY 10121. Copyright ©2021 by McGraw Hill LLC. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced or distributed in any form or by any means, or stored in a database or retrieval system, without the prior written consent of McGraw Hill LLC, including, but not limited to, in any network or other electronic storage or transmission, or broadcast for distance learning.

Some ancillaries, including electronic and print components, may not be available to customers outside the United States.

This book is printed on acid-free paper.

1 2 3 4 5 6 7 8 9 LCR 24 23 22 21 20

ISBN 978-1-26057524-8
MHID 1-260-57524-1

Cover Image: ©*Shutterstock/Brian Lasenby*

All credits appearing on page or at the end of the book are considered to be an extension of the copyright page.

The Internet addresses listed in the text were accurate at the time of publication. The inclusion of a website does not indicate an endorsement by the authors or McGraw Hill LLC, and

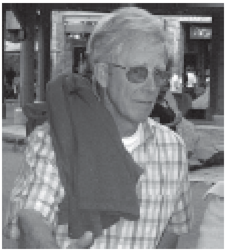
McGraw Hill LLC does not guarantee the accuracy of the information presented at these sites.

mheducation.com/highered

Dedication

—To Stan and Kate

John Dean is an Associate Professor in the Computer Science and Information Systems Department at Park University. He earned a Ph.D. degree in computer science from Nova Southeastern University and an M.S. degree in computer science from the University of Kansas. He is Java certified and has worked in industry as a software engineer and project manager, specializing in Java and various web technologies—JavaScript, JavaServer Pages, and servlets. He has taught a full range of computer science courses, including Java programming and Java-based web programming. He has authored a web programming textbook with a focus on client-side technologies HTML5, CSS, and JavaScript.



Raymond Dean is a Professor Emeritus, Electrical Engineering and Computer Science, University of Kansas. He earned an M.S. degree from MIT and a Ph.D. degree from Princeton University. As a professional engineer in the HVAC industry, he wrote computer programs that design air distribution systems and analyze energy consumption and sound propagation in buildings. At the University of Kansas, he taught microprocessor programming, data structures, and other courses in electrical engineering and computer science.

Preface x

Project Summary xxiv

Chapter **1**

Introduction to Computers and Programming 1

- **1.1 Introduction 2**
- **1.2 Hardware Terminology 2**
- **1.3 Program Development 10**
- **1.4 Source Code 12**
- **1.5 Compiling Source Code into Object Code 13**

1.6 Portability 14
1.7 Emergence of Java 15
1.8 Computer Ethics 18
1.9 First Program—Hello World 19
1.10 GUI Track: Hello World (Optional) 24

Chapter 2

Algorithms and Design 32
2.1 Introduction 32
2.2 Output 33
2.3 Variables 34
2.4 Operators and Assignment Statements 35
2.5 Input 36
2.6 Flow of Control and Flowcharts 37
2.7 if Statements 38
2.8 Loops 43

2.9 Loop Termination Techniques 45

2.10 Nested Looping 48

2.11 Tracing 51

2.12 Problem Solving: Other Pseudocode Formats and an Asset Management Example 55

Chapter 3

Java Basics 65

3.1 Introduction 66

3.2 “I Have a Dream” Program 66

3.3 Comments and Readability 67

3.4 The Class Heading 69

3.5 The main Method’s Heading 69

3.6 Braces 70

3.7 System.out.println 71

3.8 Compilation and Execution 73
3.9 Identifiers 73
3.10 Variables 74
3.11 Assignment Statements 75
3.12 Initialization Statements 77
3.13 Numeric Data Types—int, long, float, double 78
3.14 Constants 80
3.15 Arithmetic Operators 83
3.16 Expression Evaluation and Operator Precedence 86
3.17 More Operators: Increment, Decrement, and Compound Assignment 88
3.18 Tracing 90
3.19 Type Casting 90
3.20 char Type and Escape Sequences 93
3.21 Primitive Variables

Versus Reference Variables 95

3.22 Strings 96

3.23 Input—the Scanner Class 100

3.24 Simple File Input for Repetitive Testing During Program Development 105

3.25 GUI Track: Input and Output with Dialog Boxes (Optional) 107

Chapter 4

Control Statements 119

4.1 Introduction 120

4.2 Conditions and Boolean Values 120

4.3 if Statements 121

4.4 && Logical Operator 124

4.5 | | Logical Operator 129

4.6 ! Logical Operator 131
4.7 Switching Constructs 132
4.8 while Loop 138
4.9 do Loop 142
4.10 for Loop 144
4.11 Solving the Problem of Which Loop to Use 149
4.12 Nested Loops 150
4.13 boolean Variables 152
4.14 Input Validation 156
4.15 Problem Solving with Boolean Logic (Optional) 157

Chapter 5

Using Prebuilt Methods 170
5.1 Introduction 170
5.2 The API Library 171
5.3 Math Class 177
5.4 Wrapper Classes for Primitive Types 182

5.5 Character Class 186

5.6 String Methods 188

5.7 Formatted Output with the printf> Method 194

5.8 Problem Solving with Random Numbers (Optional) 199

5.9 GUI Track: Covering an Image with a Tinted Pane (Optional) 203

Interlude 213

Multiple-Method Programs in a Non-Object-Oriented Environment 213

GUI Track: Multiple-Method Program That Uses StackPane and Group to Display Images, Rectangles, Lines, an Oval, and Text (Optional) 216

Chapter 6

Object-Oriented Programming 222

6.1 Introduction 223

6.2 Object-Oriented Programming Overview 223

6.3 First OOP Class 227

6.4 Driver Class 230

6.5 Calling Object, this Reference 234

6.6 Instance Variables 236

6.7 Tracing an OOP Program 237

6.8 UML Class Diagrams 242

6.9 Local Variables 244

6.10 The return Statement 247

6.11 Argument Passing 249

6.12 Specialized Methods—Accessors, Mutators, and Boolean Methods 252

6.13 Problem Solving with

Simulation (Optional) 255

Chapter 7

Object-Oriented Programming—Additional Details 272

7.1 Introduction 273
7.2 Object Creation—A Detailed Analysis 273
7.3 Assigning a Reference 275
7.4 Testing Objects for Equality 279
7.5 Passing References as Arguments 284
7.6 Method-Call Chaining 286
7.7 Overloaded Methods 289
7.8 Constructors 293
7.9 Overloaded Constructors 299
7.10 Static Variables 303

7.11 Static Methods 306

7.12 Named Constants 312

7.13 Problem Solving with Multiple Driven Classes 314

Chapter 8

Software Engineering 324

8.1 Introduction 325

8.2 Coding-Style Conventions 325

8.3 Documentation for Outsiders 334

8.4 Helper Methods 338

8.5 Encapsulation (with Instance Variables and Local Variables) 342

8.6 Recognizing the User’s Point of View 344

8.7 Design Philosophy 345

8.8 Top-Down Design 350

8.9 Bottom-Up Design 359
8.10 Case-Based Design 361
8.11 Iterative Enhancement 361
8.12 Merging the Driver Method into the Driven Class 363
8.13 Accessing Instance Variables Without Using this 365
8.14 Writing Your Own Utility Class 366
8.15 Problem Solving with the API Calendar Class (Optional) 368
8.16 GUI Track: Problem Solving with CRC Cards (Optional) 370

Chapter 9

Arrays 384

9.1 Introduction 385
9.2 Array Basics 385
9.3 Array Declaration and Creation 387
9.4 Array length Property and Partially Filled Arrays 391
9.5 Copying an Array 393
9.6 Problem Solving with Array Case Studies 397
9.7 Searching an Array 403
9.8 Sorting an Array 408
9.9 Two-Dimensional Arrays 412
9.10 Arrays of Objects 418
9.11 For-Each Loops 425

Chapter 10

ArrayLists and an Introduction to the Java Collections

Framework 435
10.1 Introduction 436
10.2 The ArrayList Class 437
10.3 Storing Primitives in an ArrayList 443
10.4 ArrayList Example Using Anonymous Objects and the For-Each Loop 446
10.5 ArrayLists Versus Standard Arrays 450
10.6 The LinkedList Class 451
10.7 The List Interface 452
10.8 Problem Solving: How to Compare Method Execution Times 453
10.9 Queues, Stacks, and the ArrayDeque Class 457
10.10 Overview of the Java Collections Framework 464
10.11 Collections Example—Information Flow in a

Network of Friends 468
10.12 GUI Track: Second Iteration of Problem Solving with CRC Cards (Optional) 476

Chapter 11

Recursion 489

11.1 Introduction 490
11.2 Guidelines for Writing a Recursive Method 491
11.3 A Recursive Factorial Method 492
11.4 Comparison of Recursive and Iterative Solutions 496
11.5 Recursive Method Evaluation Practice 500
11.6 Binary Search 503
11.7 Merge Sort 506
11.8 Towers of Hanoi 510

11.9 Problem Solving with Performance Analysis 514

11.10 GUI Track: Drawing Trees with a Fractal Algorithm (Optional) 517

Chapter 12

Type Details and Alternative Coding Mechanisms 530

12.1 Introduction 531

12.2 Integer Types and Floating-Point Types 532

12.3 char Type and the ASCII Character Set 536

12.4 Type Conversions 538

12.5 Prefix/Postfix Modes for Increment/ Decrement Operators 541

12.6 Embedded Assignments 544

12.7 Conditional Operator

Expressions 546

12.8 Expression Evaluation Review 547

12.9 Short-Circuit Evaluation 551

12.10 Empty Statement 552

12.11 Using break to Exit from a Loop 554

12.12 for Loop Header Details 555

12.13 Enumerated Types 557

12.14 forEach Method, Lambda Expressions, Method References, and Streams 564

12.15 Hexadecimal, Octal, and Binary Numbers 573

12.16 GUI Track: Unicode (Optional) 574

12.17 Introduction to GridWorld Case Study (Optional) 579

Chapter 13

Aggregation, Composition, and Inheritance 591

13.1 Introduction 592

13.2 Composition and Aggregation 592

13.3 Inheritance Overview 599

13.4 Implementation of a Person/Employee/FullTime Hierarchy 603

13.5 Constructors in a Subclass 605

13.6 Method Overriding 606

13.7 Using the Person/Employee/FullTime Hierarchy 609

13.8 The final Access Modifier 610

13.9 Using Inheritance with Aggregation and

Composition 610
13.10 Design Practice with Card Game Example 613
13.11 GridWorld Case Study Extensions (Optional) 619
13.12 Problem Solving with Association Classes (Optional) 626

Chapter 14

Inheritance and Polymorphism 637
14.1 Introduction 638
14.2 The Object Class and Automatic Type Promotion 638
14.3 The equals Method 639
14.4 The toString Method 643
14.5 Polymorphism and Dynamic Binding 648

14.6 Assignments When the Two Sides' Classes Are Different 653

14.7 Polymorphism with Arrays 654

14.8 abstract Methods and Classes 660

14.9 Interfaces 663

14.10 The protected Access Modifier 673

14.11 GUI Track: Three-Dimensional Graphics (Optional) 677

Chapter 15

Exception Handling 691

15.1 Introduction 692

15.2 Overview of Exceptions and Exception Messages 692

15.3 Using try and catch Blocks to Handle

"Dangerous" Method Calls 693
15.4 Line Plot Example 695
15.5 try Block Details 699
15.6 Two Categories of Exceptions—Checked and Unchecked 700
15.7 Unchecked Exceptions 702
15.8 Checked Exceptions 705
15.9 Generic catch Block with Exception Class 708
15.10 Multiple catch Blocks and Multiple Exceptions per Block 712
15.11 Understanding Exception Messages 714
15.12 Using a throws Clause to Postpone the catch 718
15.13 Automatic Cleanup Using Try-With-Resources 720

15.14 GUI Track: Line Plot Example Revisited (Optional) 722

Chapter 16

Files, Buffers, Channels, and Paths 735

16.1 Introduction 736

16.2 Simple Text-File Example: HTML File Generator 737

16.3 A Website Reader 741

16.4 Object File I/O 743

16.5 Character Sets and File-Access Options 748

16.6 Buffered Text File I/O 749

16.7 Primitive Buffers with Random Access 752

16.8 Channel I/O and Memory-Mapped Files 760

16.9 Path, Whole-File, and Directory Operations 767

16.10 Walking a Directory Tree 769

16.11 GUI Track: Final Iteration of Problem Solving with CRC Cards (Optional) 775

Chapter 17

GUI Programming Basics 787

17.1 Introduction 788

17.2 SimpleWindow Program 791

17.3 Stage and Scene 794

17.4 JavaFX Components 796

17.5 Label Control 797

17.6 TextField Control 799

17.7 Greeting Program 801

17.8 Event Handling 805

17.9 Property Binding 809

17.10 JavaFX CSS 812

17.11 Scene Graph Inheritance 818

17.12 Style Sheets and Cascading 821

17.13 Button Control and FactorialButton Program 826

17.14 Distinguishing Between Multiple Events 832

17.15 Colors 834

17.16 ColorChooser Program 838

Chapter 18

GUI Programming—Layout Panes 849

18.1 Introduction 849

18.2 Layout Panes 851

18.3 FlowPane and GridPane—Competing Layout Philosophies 853

18.4 VBox Program with Two Stages and an Image File 858

18.5 BorderPane 867

18.6 TilePane and TextFlow Containers 872

18.7 TicTacToe Program 878

18.8 Embedded Panes, HBox, and MathCalculator Program 882

18.9 Plain Pane Container and Component Positioning 889

Chapter 19

GUI Programming—Additional GUI Components, Additional Event Handlers, Animation 896

19.1 Introduction 897

19.2 User Interface Design 897

19.3 TextArea Control 898

19.4 CheckBox and RadioButton Controls 903

19.5 ComboBox Control 909

19.6 Job Application Program 913

19.7 ScrollPane and Menu Classes 918

19.8 Images and Mouse Events 922

19.9 Lunar Eclipse Program with Circle, RadialGradient, and Slider 928

19.10 Animation 933

Appendices

Appendix 1 ASCII Character Set 943

Appendix 2 Operator

Precedence 945

Appendix 3 Java Keywords and Other Reserved Words 947

Appendix 4 Packages and Modules 951

Appendix 5 Java Coding-Style Conventions 963

Appendix 6 Javadoc with Tags 975

Appendix 7 UML Diagrams 980

Appendix 8 Number Systems and Conversions Between Them 986

Additional Online Material

Chapter S6 Writing Methods in a Non-Object-Oriented Environment

Chapter S9 Arrays in a Non-Object-Oriented Environment

Chapter S17 GUI Programming Basics (with Swing and AWT)

Chapter S18 GUI Programming—Component Layout, Additional GUI Components (with Swing and AWT)

Appendix 9 Multithreading

Index 990

McGraw
Hill
Education

5011111

In this book, we lead you on a journey into the fun and exciting world of computer programming. Throughout your journey, we'll provide you with lots of problem-solving practice. After all, good programmers need to be good problem solvers. We'll show you how to implement your problem solutions with Java programs. We provide a plethora of examples, some short and focused on a single concept, some longer and more "real world." We present the material in a conversational, easy-to-follow manner aimed at making your journey a pleasant one. When you're done with the book, you should be a proficient Java programmer.

Our textbook targets a wide range of readers. Primarily, it targets students in a standard college-level "Introduction to Programming" course or course sequence where no prerequisite programming experience is assumed. We have included the topics recommended by the College Board for high school students studying for advanced placement (AP) in computer science. So this text should be good for those students as well.

In addition to targeting students with no prerequisite programming experience, our textbook targets industry practitioners and college-level students who have some programming experience and want to learn Java. This second set of readers can skip the early chapters on general programming concepts and focus on the features of

Java that differ from the languages that they already know. In particular, because C++ and Java are similar, readers with a C++ background should be able to cover the textbook in a single three-credit-hour course. (But we should reiterate for those of you with no programming experience: No prerequisite programming experience is required in order to use this text.)

Finally, our textbook targets those who are learning Java on their own, outside of a classroom environment. This third set of readers should read the entire textbook at a pace determined on a case-by-case basis.

What's New in This Edition?

The changes in this edition are big and small. Big changes include new chapters, reorganized chapter sections, new programming constructs, new program examples, and new exercises. Smaller changes include updating explanations and anecdotes. We've combed the entire book for opportunities to improve the book's clarity and readability. The following list highlights the more significant changes that we've made to this edition.

- **Introductory Chapter**

  To keep up with the computer industry's growth, we've made quite a few changes to Chapter 1, such as updating the information in the computer hardware and Java history sections.

- **Switching Constructs**

  Java 12 and Java 13 made improvements to the venerable switch statement, and this edition describes those improvements. We use the new switching techniques (multiple comma-separated case constants and no break statements) for programs throughout the book. And we use switch statements versus switch expressions according to what the problem calls for. If you’re a fan of the old-style switch statement, no worries, we provide a description early on which will help you with legacy code.

- **Local Variable Type Inferencing**

Java 10 introduced the ability to use var as a type (rather than int, double, etc.) for a local variable declaration where the declaration is part of an initialization. We describe the new syntax, but for self-documentation reasons, we stick with traditional explicit type declarations for the most part.

- **Name Change for Static

Variables and Static Methods

The powers that be (the Oracle documentation folks) now use the terms *static variable* and *static method* for what used to be known as class variable and class method, so we've updated accordingly.

- **Miscellaneous Java API Library Updates**

With the new Java releases since the second edition, there have been quite a few updates to the Java API library. We've updated our discussions and programs with new API method and constructor calls when appropriate. Most of our new API content can be found in our GUI coverage, but there are other API changes sprinkled throughout the book. For example, with Java's deprecation of the wrapper class constructors, we've refactored our programs to rely on the wrapper classes’ valueOf methods.

- **New Section—**forEach **Method and Streams**

We introduce the forEach method as a simple alternative to the for-each loop in the context of an ArrayList. We then use the forEach method in the context of streams, where it really shines. We describe streams in depth, with their exciting potential to take advantage of parallel processing to improve a program’s efficiency.

- **Lambda Expressions and Method References**

Lambda expressions and method references are techniques that allow you to implement the functionality of a method so you can use it as an argument in a method call. We first present lambda expressions and method references as arguments for a forEach method call. Later, we use lambda expressions and method references extensively to help with the GUI programs.

- **Interfaces with Static Methods and Default Methods**

We've rewritten Chapter 14's section on interfaces to include a discussion of static methods and default methods. Oracle added them to interfaces because they support an interface's ability to implement multiple inheritance effectively.

- **End-of-Chapter GUI Sections**

We've rewritten all of our end-of-chapter GUI sections to take advantage of Java's newer GUI constructs.

- **Three New Chapters—JavaFX**

In this book's second edition, we

used the AWT and Swing platforms for our two GUI chapters. This third edition moves those chapters to the book's website. We provide three new chapters in the main body of the book that describe GUI programming using the JavaFX platform. As part of that presentation, you'll learn how to format your programs using JavaFX CSS properties.

- **New Appendix—Modules**

In Appendix 4, we introduce modules, which allow you to group together packages. Modules make it easier to organize and share classes for different programming needs. They are used to facilitate the configuration of Java software for diverse hardware and software platforms.

- **New Exercises**

We have substantially changed most of the exercises and altered almost all of them in some way. As before, we provide exercise solutions on the password-protected instructor's portion of the book's website.

Compliant with the College Board's AP Computer Science A Curriculum

We have put a great deal of effort into ensuring that this textbook is compliant with the College Board's Advanced Placement (AP) Computer Science A curriculum content. It follows all the AP Computer Science A guidelines. As such, it appears on the College Board's approved textbook list at https://apcentral.collegeboard.org/courses/ap-computer-science-a/course-audit.

Textbook Cornerstone #1: Problem Solving

Being able to solve problems is a critical skill that all programmers must possess. We teach programmatic problem solving by emphasizing two of its key elements—algorithm development and program design.

Emphasis on Algorithm Development

In Chapter 2, we immerse readers into algorithm development by using pseudocode for the algorithm examples instead of Java. In using pseudocode, students are able to work through nontrivial problems on their own without getting bogged down in Java syntax—no need to worry about class headings, semicolons, braces, and so on.[1] Working through nontrivial problems enables students to gain an early appreciation for creativity, logic, and organization. Without that appreciation, Java students tend to learn Java syntax with a rote-memory attitude. But with that appreciation, students tend to learn Java syntax more quickly and effectively because they have a motivational basis for learning it. In addition, they are able to handle nontrivial Java homework assignments fairly early because they have prior experience with similarly nontrivial pseudocode homework assignments.

In Chapter 3 and in later chapters, we rely primarily on Java for algorithm-development examples. But for the more involved problems, we sometimes use high-level pseudocode to describe first-cut proposed solutions.

Using pseudocode enables readers to bypass syntax details and focus on the algorithm portion of the solution.

Emphasis on Program Design

Problem solving is more than just developing an algorithm. It also involves figuring out the best implementation for the algorithm. That's program design. Program design is extremely important, and that's why we spend so much time on it. Frequently, we explain the thought processes that a person might go through when coming up with a solution. For example, we explain how to choose between different loop types, how to split up a method into multiple methods, how to decide on appropriate classes, how to choose between instance and static members, and how to determine class relationships using inheritance and composition. We challenge students to find the most elegant implementations for a particular task.

We devote a whole chapter to program design—Chapter 8, “Software Engineering.” In that chapter, we provide an in-depth look at coding-style conventions and documentation for programmers and users. We discuss design strategies like separation of concerns, modularization, and encapsulation. Also in the chapter, we describe alternative design strategies—top-down, bottom-up, case-based, and iterative enhancement.

Problem-Solving Sections

We often address problem solving (algorithm development and program design) in the natural flow of explaining concepts. But we also cover problem solving in sections that are wholly devoted to it. In each problem-solving section, we present a situation that contains an unresolved problem. In coming up with a solution for the problem, we try to mimic the real-world problem-solving experience by using an iterative design strategy. We present a first-cut solution, analyze the solution, and then discuss possible improvements to it. We use a conversational trial-and-error format (e.g., “What type of layout manager should we use? We first tried the GridLayout manager. That works OK, but not great. Let’s now try the BorderLayout manager.”). This casual tone sets the student at ease by conveying the message that it is normal, and in fact expected, that a programmer will need to work through a problem multiple times before finding the best solution.

Additional Problem-Solving Mechanisms

We include problem-solving examples

and problem-solving advice throughout the text (not just in Chapter 2, Chapter 8, and the problem-solving sections). As a point of emphasis, we insert a problem-solving box, with an icon and a succinct tip, next to the text that contains the problem-solving example and/or advice.

We are strong believers in learning by example. As such, our textbook contains a multitude of complete program examples. Readers are encouraged to use our programs as recipes for solving similar programs on their own.

Textbook Cornerstone #2: Fundamentals First

Postpone Concepts That Require Complex Syntax

We feel that many introductory

programming textbooks jump too quickly into concepts that require complex syntax. In using complex syntax early, students get in the habit of entering code without fully understanding it or, worse yet, copying and pasting from example code without fully understanding the example code. That can lead to less-than-ideal programs and students who are limited in their ability to solve a wide variety of problems. Thus, we prefer to postpone concepts that require complex syntax. We prefer to introduce such concepts later on, when students are better able to understand them fully.

As a prime example of that philosophy, we cover the simpler forms of GUI programming early (in an optional graphics track), but we cover the more complicated forms of GUI programming later in the book. Specifically, we postpone event-driven

GUI programming until the end of the book. This is different from some other Java textbooks, which favor early full immersion into event-driven GUI programming. We feel that strategy is a mistake because proper event-driven GUI programming requires a great deal of programming maturity. When they learn it at the end of the book, our readers are better able to understand it fully.

Tracing Examples

To write code effectively, it's imperative to understand code thoroughly. We've found that step-by-step tracing of program code is an effective way to ensure thorough understanding. Thus, in the earlier parts of the textbook, when we introduce a new programming structure, we often illustrate it with a meticulous trace. The detailed tracing technique we use illustrates the thought process programmers employ while debugging. It’s a printed alternative to the sequence of screen displays generated by debuggers in integrated development environment (IDE) software.

Input and Output

In the optional GUI-track sections and in the GUI chapters at the end of the book, we use GUI commands for input and output (I/O). But because of our emphasis on fundamentals, we use console commands for I/O for the rest of the book.[2] For console input, we use the Scanner class. For console output, we use the standard System.out.print, System.out.println, and System.out.printf methods.

Textbook Cornerstone #3: Real World

More often than not, today's classroom students and industry practitioners prefer to learn with a hands-on, real-world approach. To meet this need, our textbook and its associated website include:

- compiler tools
- complete program examples
- practical guidance in program design
- coding-style guidelines based on industry standards
- Unified Modeling Language (UML) notation for class relationship diagrams
- practical homework-project assignments

Compiler Tools

We do not tie the textbook to any particular compiler tool—you are free to use any compiler tool(s) that you like. If you do not have a preferred compiler in mind, then you might want to try out one or more of these:

- Java Standard Edition Development Kit (JDK), by Oracle
- TextPad, by Helios
- Eclipse, by the Eclipse Foundation
- Netbeans, backed by Oracle
- BlueJ, by the University of Kent and Deaken University

To obtain the above compilers, visit our textbook website at http://www.mhhe.com/dean3e, find the appropriate compiler link(s), and download away for free.

Complete Program Examples

In addition to providing code fragments to illustrate specific concepts, our textbook contains lots of complete program examples. With complete programs, students are able to (1) see how the analyzed code ties in with the rest of a program, and (2) test the code by running it.

Coding-Style Conventions

We include coding-style tips throughout the textbook. The coding-style tips are based on Oracle's coding conventions (https://www.oracle.com/technetwork/java/codeconvtoc-136057.html), Google's coding conventions (https://google.github.io/styleguide/javaguide.html), and industry practice. In Appendix 5, we provide a complete reference for the book's coding-style conventions and an associated example program that illustrates these conventions.

UML Notation

UML has become a standard for describing the entities in large software projects. Rather than overwhelm beginning programmers with syntax for the entire UML (which is quite extensive), we present a subset of UML. Throughout the textbook, we incorporate UML notation to represent classes and class relationships pictorially. For those interested in more details, we provide additional UML notation in Appendix 7.

Homework Problems

We provide homework problems that are illustrative, practical, and clearly worded. The problems range from easy to challenging. They are grouped into three categories—review questions, exercises, and projects. We include review questions and exercises at the end of each chapter, and we provide projects on our textbook's website.

The review questions tend to have short answers, and the answers are in the textbook. The review questions use these formats: short-answer, multiple-choice, true/false, fill-in-the-blank, tracing, debugging, and write a code fragment. Each review question is based on a relatively small part of the chapter.

The exercises tend to have short to moderate-length answers, and the answers are not in the textbook. The exercises use these formats: short-answer, tracing, debugging, and write a code fragment. Exercises are keyed to the highest prerequisite section number in the chapter, but they sometimes integrate concepts from several parts of the chapter. For this third edition, we have changed almost all of the end-of-chapter exercises, including exercises associated with unchanged material in the body of the text.

The projects consist of problem descriptions whose solutions are complete programs. Project solutions are not in the textbook. Projects require students to employ creativity and problem-solving skills and apply what they've learned in the chapter. These projects often include optional parts, which provide challenges for the more talented students. Projects are keyed to the highest prerequisite section number in the chapter, but they often integrate concepts from several preceding parts of the chapter. For this third edition, we have modified old projects and added new projects to make all projects conform to content in the body of the current text. Because the most substantial body-of-text changes are in the final three chapters, most of the project modifications and additions are associated with these chapters.

An important special feature of this book is the way that it specifies problems. “Sample sessions” show the precise output generated for a particular set of input values. These sample sessions include inputs that represent typical situations and sometimes also extreme or boundary situations.

Academic-Area Projects

To enhance the appeal of projects and to show how the current chapter’s programming techniques might apply to different areas of interest, we take project content from several academic areas:

- computer science and numerical methods
- business and accounting
- social sciences and statistics
- math and physics

- engineering and architecture
- biology and ecology

Most of the academic-area projects do not require prerequisite knowledge in a particular area. Thus, instructors are free to assign almost any of the projects to any of their students. To provide a general reader with enough specialized knowledge to work a problem in a particular academic area, we sometimes expand the problem statement to explain a few special concepts in that academic area.

Most of the academic-area projects do not require students to have completed projects from earlier chapters; that is, most projects do not build on previous projects. Thus, for the most part, instructors are free to assign projects without worrying about prerequisite projects. In some cases, a project repeats a previous chapter’s project with a different approach. The teacher may elect to take advantage of this repetition to dramatize the availability of alternatives, but this is not necessary.

Project assignments can be tailored to fit readers' needs. For example:

- For readers outside of academia —

  Readers can choose projects that match their interests.

- When a course has students from one academic area—

  Instructors can assign projects from the relevant academic area.

- When a course has students with diverse backgrounds—

  Instructors can ask students to choose projects from their own academic areas, or instructors can ignore the academic-area delineations and simply assign projects that are most appealing.

To help you decide which projects to work on, we've included a "Project Summary" section after the preface. It lists all the projects by chapter and section, and for each project, it specifies:

- prerequisite chapter and section
- academic area
- estimated difficulty
- a title and brief description

After using the "Project Summary" section to get an idea of which projects you might like to work on, see the textbook's website for the full project descriptions.

Organization

In writing this book, we lead readers through three important programming methodologies: structured programming, OOP, and event-driven programming. For our structured programming coverage, we introduce basic concepts such as variables and operators, if statements, and loops. Then we show readers how to call prebuilt methods from Oracle’s Java API library. Many of these methods, like those in the Math class, are non-OOP methods that can be called directly. Others, like those in the String class, are OOP methods that must be called by a previously created object. After an “interlude” that gives readers a brief taste of what it’s like to write methods in a non-OOP environment, we move into OOP programming, and introduce basic OOP concepts such as classes, objects, instance variables, instance methods, and constructors. We also introduce static variables and static methods, which are useful in certain situations. However, we note that they should be used less often than instance variables and instance methods. Next, we move on to more advanced OOP concepts—arrays, collections, interfaces, and inheritance. Chapters on exception handling and files provide a transition into event-driven GUI programming. We describe and employ event-driven GUI programming in the final three chapters.

The content and sequence we promote enable students to develop their skills from a solid foundation of programming fundamentals. To foster this fundamentals-first approach, our book starts with a minimum set of concepts and details. It then gradually broadens concepts and adds detail later. We avoid overloading early chapters by deferring certain less-important details to later chapters.

GUI Track

Many programmers find Graphical User Interface (GUI) programming to be fun. As such, GUI programming can be a great motivational tool for keeping readers interested and engaged. That's why we include graphics sections throughout the book, starting in Chapter 1. We call those sections our "GUI track." Most of these end-of-chapter sections use GUI code that complements the current chapter's previously presented non-GUI material. For readers who do not have time for the GUI track, no problem. Any or all of the GUI track sections may be skipped because the rest of the book does not depend on any of the GUI-track material.

Although the rest of the book does not depend on the GUI-track material, be aware that some of the GUI-track sections depend on some of the material in prior GUI-track sections:

- Chapter 3's GUI section introduces dialog boxes for user input, and dialog boxes are used in later GUI sections for Chapters 10, 11, 15, and 16.
- Chapters 8, 10, and 16 have GUI sections that implement a common program, with iterative enhancements in each new GUI section.
- Chapters 12 and 13 have GUI sections that implement a common GridWorld program (for readers interested in the College Board's AP Computer Science A curriculum). The GridWorld code uses AWT and Swing GUI software.

Chapter 1

In Chapter 1, we first explain basic computer terms—what are the hardware components, what is source code, what is object code, and so on. We then narrow our focus and describe the programming language we'll be using for the remainder of the book—Java. Finally, we give students a quick view of the classic bare-bones "Hello World" program. We explain how to create and run the program using minimalist software—Microsoft's Notepad text editor and Oracle's command-line JDK tools.

Chapter 2

In Chapter 2, we present problem-solving techniques with an emphasis on algorithmic design. In implementing algorithm solutions, we use generic tools—flowcharts and pseudocode—with pseudocode given greater weight. As part of our algorithm-design explanation, we describe structured programming techniques. In order to give students an appreciation for semantic details, we show how to trace algorithms.

Chapters 3–5

We present structured programming techniques using Java in Chapters 3–5. Chapter 3 describes sequential programming basics—variables, input/output, assignment statements, and simple method calls. Chapter 4 describes nonsequential program flow—if statements, switch constructs, and loops. In Chapter 5, we explain methods in more detail and show readers how to use prebuilt methods in the Java API library. In all three chapters, we teach algorithm design by solving problems and writing programs with the newly introduced Java syntax.

Interlude

This "mini-chapter" contains two programs that show how to write multiple methods without using OOP.

The Interlude presents a fork in the road between two study sequences. For the standard study sequence, read the chapters in the standard order (Chapters 1 through 19). For the “objects later” study sequence, after reading Chapter 5, read the supplemental chapters S6 and S9 online before returning to Chapter 6, where you’ll begin your study of OOP in earnest.

Chapters 6–7

Chapter 6 introduces the basic elements of OOP in Java. This includes implementing classes and implementing methods and variables within those classes. We use UML class diagrams and object-oriented tracing techniques to illustrate these concepts.

Chapter 7 provides additional OOP details. It explains how reference variables are assigned, tested for equality, and passed as arguments to a method. It explains overloaded methods and constructors. It also explains the use of static variables, static methods, and different types of named constants.

Chapter 8

While the art of program design and the science of computerized problem-solving are developed throughout the textbook, in Chapter 8, we focus on these aspects in the context of OOP. This chapter begins with an organized treatment of programming style. It introduces javadoc, the Java application that automatically generates documentation for user-programmers. It describes ways to communicate with users who are not programmers. It describes organizational strategies like separation of concerns, modularization, encapsulation, and provision of general-purpose utilities. Coded examples show how to implement these strategies. It describes the major programming paradigms—top-down design, bottom-up design, using pre-written software for low-level modules, and prototyping.

Chapters 9–10

Chapter 9 describes arrays, including arrays of primitives, arrays of objects, and multidimensional arrays. It illustrates array use with complete programs that sort, search, and construct histograms. Chapter 10 describes Java's powerful array alternative, ArrayList. This provides a simple example of generic-element specification. It also introduces the Java Collections Framework, which in turn, provides natural illustrations of Java interfaces. The prewritten classes in the Java Collections Framework provide a simple introduction of sets, maps, and queues. A relatively short but complete program shows how the pre-written Java implementations of these data structures can be used to create and traverse a multiconnected random network.

Chapter 11

Chapter 11 describes another way to process a collection of data—recursion. This chapter includes a discussion of various recursive strategies. It introduces recursion with a real-life example and a familiar problem that one can solve easily with either looping or recursion. Then it moves gradually to problems that are harder to solve with looping and more easily solved with recursion. Although this chapter appears after the chapter on ArrayLists and the Java Collections Framework, it does not depend on these concepts—it uses just ordinary arrays.

Chapter 12

Early on, students need to be immersed in problem-solving activities. Covering too much syntax detail early can detract from that objective. Thus, we initially gloss over some less-important syntax details and come back to those details later in Chapter 12. This chapter provides more details on items such as these:

- byte and short primitive types
- Unicode character set
- type promotions
- postfix versus prefix modes for the increment and decrement operators
- conditional operator
- short-circuit evaluation
- enum data type

- forEach method
- lambda expressions
- method references
- streams

The chapter ends with a friendly introduction to a relatively large public-domain program called GridWorld, which the College Board has used for many years as part of its AP Computer Science A course of study. This gives students a glimpse of how larger programs are organized.

Chapters 13–14

Chapters 13 and 14 describe class relationships in depth with numerous examples. Chapter 13 describes aggregation, composition, and inheritance. Chapter 14 describes advanced inheritance-related details such as the Object class, polymorphism, abstract classes, and the finer points of interfaces. An optional section at the end of Chapter 13 describes an extension of the GridWorld environment introduced in Chapter 12 and provides additional exposure to Java's legacy AWT and Swing graphics. Exercises in Chapters 13 and 14 relate material in these two chapters to corresponding GridWorld features.

Chapters 15 – 16

Chapter 15 describes exception handling, and Chapter 16 describes files. We present exception handling before files because file-handling code requires the use of exception handling. For example, to open a file one must check for an exception. In addition to simple text I/O, our treatment of files includes buffering, random access, channeling, and memory mapping.

Chapters 17 – 19

As in the end-of-chapter GUI sections, Chapters 17 – 19 present GUI concepts using the JavaFX platform. But the programming strategies differ. What follows are the strategies used in Chapters 17–19 (which are different from the strategies used in the end-of-chapter GUI sections). For user input, the programs use the components TextField, TextArea, Button, RadioButton, CheckBox, ComboBox, ScrollPane, and Menu. For layout, the programs use the containers FlowPane, VBox, HBox, GridPane, BorderPane, TilePane, and TextFlow. For formatting, the programs use JavaFX CSS properties.

Depending on how much and what kind of GUI techniques you're interested in, you can study one or more of the end-of-chapter GUI sections or skip all of them. That won't affect your ability to grasp what's in Chapters 17 – 19. If you're short on time, you can omit all of the book's GUI material without compromising your understanding of other material in the book.

Chapters S17 – S18

Chapters S17 and S18 (the *S*'s stand for supplemental) are posted online. They describe the older Java GUI platforms —AWT and Swing. The trend has been for new Java programs to use JavaFX, and not AWT and Swing. However, there's quite a bit of AWT and Swing code currently in production, and that means there's still a need for Java programmers to understand the older techniques so they can update and improve existing code. So if you find yourself in that position, Chapters S17 and S18 are a good starting point for learning what you need to know.

Appendices

Most of the appendices cover reference material, like the ASCII character set and the operator precedence table. For this third edition, we have updated Appendix 4 by including a detailed description of Java modules.

Subject-Matter Dependencies and Sequence-Changing Opportunities

We've positioned the textbook's material in a natural order for someone who wants fundamentals and also wants an early introduction to OOP. We feel that our order is the most efficient and effective one for learning how to become a proficient OOP programmer. Nonetheless, we realize that different readers have different content-ordering preferences. To accommodate those different preferences, we’ve provided some built-in flexibility. Figure 0.1 illustrates that flexibility by showing chapter dependencies and, more importantly, chapter nondependencies. For example, the arrow between Chapter 3 and Chapter 4 means that Chapter 3 must be read prior to Chapter 4. Because there are no arrows going out of Chapters 1, 11, and 16 that point to other complete chapters, you may skip those chapters without losing prerequisite material that later chapters need. We use rectangles with rounded corners to indicate chapter sections that you may want to read in advance. If you choose that option, you’ll want to return to the normal chapter sequence after completing the advanced sections.

Here are some sequence-changing opportunities revealed by Figure 0.1:

- Readers can skip Chapter 1,

“Introduction to Computers and Programming.”

- For an earlier introduction to OOP, readers can read the OOP overview section in Chapter 6 after reading Chapter 1.

- Readers can learn OOP syntax and semantics in Chapter 6 after finishing Java basics in Chapter 3.

- For additional looping practice, readers can learn about arrays in Chapter 9 after finishing loops in Chapter 4.

- Readers can skip Chapter 11, “Recursion,” and Chapter 16, “Files.”

- Readers who prefer a late objects approach can postpone reading Chapter 6, “Object-Oriented Programming,” by first reading Chapter S6, “Writing Methods in a Non-Object-Oriented

Environment," Sections 9.1–9.6 (which explain the basics of arrays), and Chapter S9, "Arrays in a Non-Object- Oriented Environment."

- For GUI programming, readers who prefer the Swing platform should read Chapters S17 and S18.

To support content-ordering flexibility, the book contains "hyperlinks." A hyperlink is an optional jump forward from one place in the book to another place. The jumps are legal in terms of prerequisite knowledge, meaning that the jumped-over (skipped) material is unnecessary for an understanding of the later material. We supply hyperlinks for each of the nonsequential arrows in Figure 0.1. For example, we supply hyperlinks that go from Chapter 1 to Chapter 6 and from Chapter 3 to Chapter 12. For each hyperlink tail end (in the earlier chapter), we tell the reader where they may optionally jump to. For each hyperlink target end (in the later chapter), we provide an icon at the side of the target text that helps readers find the place where they are to begin reading.

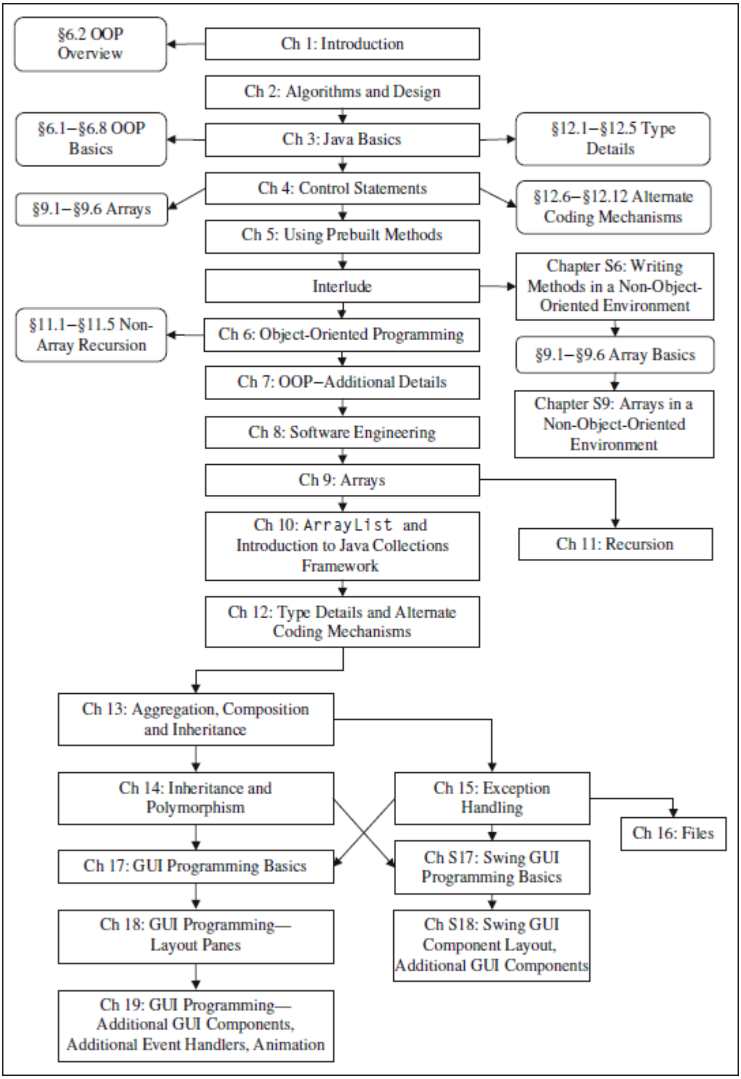

§6.2 OOP Overview
Ch 1: Introduction
Ch 2: Algorithms and Design
§6.1–§6.8 OOP Basics
Ch 3: Java Basics
§12.1–§12.5 Type Details
§9.1–§9.6 Arrays
Ch 4: Control Statements
§12.6–§12.12 Alternate Coding Mechanisms
Ch 5: Using Prebuilt Methods
Interlude
Chapter S6: Writing Methods in a Non-Object-Oriented Environment
§11.1–§11.5 Non-Array Recursion
Ch 6: Object-Oriented Programming
§9.1–§9.6 Array Basics
Ch 7: OOP–Additional Details
Chapter S9: Arrays in a Non-Object-Oriented Environment
Ch 8: Software Engineering
Ch 9: Arrays
Ch 10: ArrayList and Introduction to Java Collections Framework
Ch 11: Recursion
Ch 12: Type Details and Alternate Coding Mechanisms
Ch 13: Aggregation, Composition and Inheritance
Ch 14: Inheritance and Polymorphism
Ch 15: Exception Handling
Ch 16: Files
Ch 17: GUI Programming Basics
Ch S17: Swing GUI Programming Basics
Ch 18: GUI Programming—Layout Panes
Ch S18: Swing GUI Component Layout, Additional GUI Components
Ch 19: GUI Programming—Additional GUI Components, Additional Event Handlers, Animation

Figure 0.1 Chapter dependencies

Page xxii

Pedagogy

Icons

Program elegance.

Indicates that the associated text deals with a program's coding style, readability, maintainability, robustness, and scalability. Those qualities comprise a program's elegance.

Problem solving.

Indicates that the associated text deals with problem-solving issues. Comments associated with this icon attempt to generalize highlighted material in the adjacent text.

⚠Common errors.

Indicates that the associated text deals with common errors.

Hyperlink target.

Indicates the target end of a

hyperlink.

Program efficiency.

Indicates that the associated text refers to program-efficiency issues.

Student Resources

At the textbook website, http://www.mhhe.com/dean3e, students (and also teachers) can view and download these resources:

- Links to compiler software—for Oracle's JDK, Helios's TextPad, Eclipse, NetBeans, and BlueJ
- TextPad tutorial
- Eclipse tutorials
- Student-version Microsoft PowerPoint lecture slides without hidden notes
    - The student-version slides are identical to the teacher-version

slides except that the hidden notes, hidden slides, and quizzes are omitted.

  - ○ Omitting the hidden notes forces the students to go to lecture to hear the sage on the stage fill in the blanks. ☺

- GridWorld code
- Project assignments
- All textbook example programs and associated resource files
- Supplemental chapters
- Supplemental appendices

Instructor Resources

At the textbook website, http://www.mhhe.com/dean3e, instructors can view and download these resources:

- Teacher-version PowerPoint

lecture slides with hidden notes

  - Hidden notes provide comments that supplement the displayed text in the lecture slides.
  - For example, if the displayed text asks a question, the hidden notes provide the answer.

- Exercise solutions
- Project solutions
- Test bank materials

Acknowledgments

The inspiration for this book is teaching, without which there would be no book. So thanks to the students who have used prior editions of the textbook and provided valuable feedback. What helps them to learn is what drives improvements in the book.

And also a big thanks to the teachers who have used the book. With their diverse views from diverse backgrounds, we're able to glean important insights that can help make the book approachable to a wider audience.

Anyone who has written a textbook can attest to what a large and well-orchestrated team effort it requires. Such a book can never be the work of only one person, or even a few people. We are deeply indebted to the team at McGraw-Hill Higher Education who have shown continued faith in our writing and invested generously in it.

It has been a pleasure to work with Product Developer Heather Ervolino, who has overseen this third edition's development process. Helping us through the various stages of this edition's production was Project Manager Jason Stauter. We would also like to thank Senior Marketing

Manager Shannon O'Donnell, who helped in the final stages.

All the professionals we have encountered throughout the McGraw-Hill organization have been wonderful to work with, and we sincerely appreciate their efforts.

We would like to acknowledge with appreciation the numerous and valuable comments, suggestions, and constructive criticisms, and praise from the many instructors who have reviewed this edition and prior editions of the book. In particular,

Christopher Crick, *Oklahoma State University*

Christopher Taylor, *Milwaukee School of Engineering*

Chris Johnson, *University of Wisconsin, Eau Claire*

Dennis Lang, *Kansas State University*

Daniel Leyzberg, *Princeton University*

Geoffrey Decker, *Northern Illinois University*

Jeffrey A. Meunier, *University of Connecticut*

Mark Pauley, *University of Nebraska at Omaha*

Patricia Roth Pierce, *Southern Polytechnic State University*

We would like to thank Park University colleagues Guillermo Tonsmann and Ken Dewey for their ongoing support. They are dedicated to providing a solid foundation to their students in terms of problem solving, object-oriented programming, and Java fundamentals, and it shows.

Thanks to the students who provided feedback and searched diligently for mistakes in order to earn bonus points on their homework. In particular, thank you to Shyan Locke, Michael Dake, Marcus Shivers, Olivia Leung, Manish Lamsal, Pranoj Thapa, Peter Boyle, and Jake Thomas.

Sincerely,

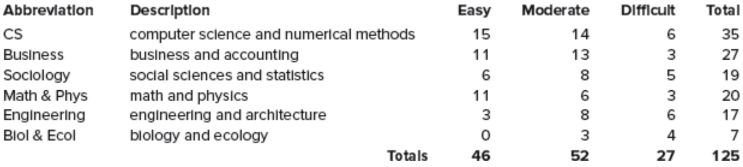
John and Ray

---

[1] Inevitably, we use a particular style for our pseudocode, but we repeatedly emphasize that other pseudocode styles are fine so long as they convey the intended meaning. Our pseudocode style is a combination of free-form description for high-level tasks and more specific commands for low-level tasks. We've chosen a pseudocode style that is intuitive, to welcome new programmers, and structured, to accommodate program logic.

[2] We introduce GUI I/O at the end of Chapter 3 with dialog boxes. That opens up an optional door for GUI fans. If readers are so inclined, they can use the Alert, TextInputDialog, and ChoiceDialog classes to implement all our programs with GUI I/O rather than console I/O.

One of the special features of this text is the diversity of its projects. Project subject matter spans six broad academic areas, as this short table shows:



Abbreviation	Description	Easy	Moderate	Difficult	Total
CS	computer science and numerical methods	15	14	6	35
Business	business and accounting	11	13	3	27
Sociology	social sciences and statistics	6	8	5	19
Math & Phys	math and physics	11	6	3	20
Engineering	engineering and architecture	3	8	6	17
Biol & Ecol	biology and ecology	0	3	4	7
	Totals	**46**	**52**	**27**	**125**

The abbreviation in the first column above will be used in a larger table below as a brief identification of a particular academic area. The four right-side columns in the above table indicate the number of projects in various categories. Of course, the highest number of projects (35) occurs in the area of computer science and numerical methods. The 29 easy and moderate CS projects are typical CS introductory programming problems. The 6 difficult CS projects provide gentle introductions to some advanced topics like linked list operations, database operations, and simulated annealing.

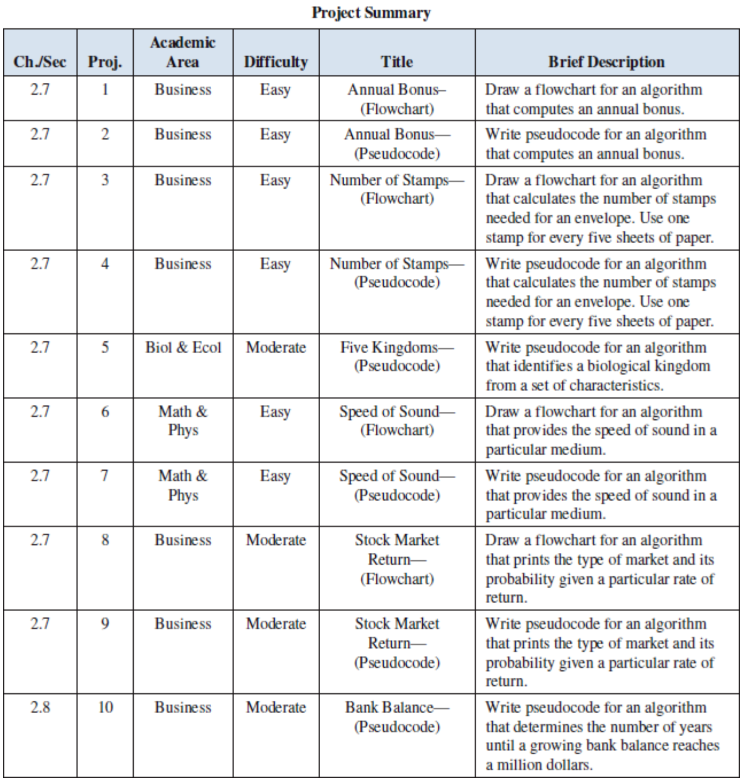
In addition, there are 27 projects in business and accounting, which include miscellaneous financial calculations, simple bookkeeping problems, and cost-accounting applications. There are 19 projects in social sciences and statistics, which include applications in sociology and political science, as well as general experience. There are 20 projects in math and physics, which include applications in both classical and chaotic mechanics. There are 17 projects in engineering and architecture, which include applications in heating ventilating and air conditioning (HVAC), electrical engineering, and civil engineering. Finally, there are 7 projects in biology and ecology, which include realistic growth and predator-prey simulations. Although we've associated each project with one primary academic area, many of these projects can fit into other academic areas as well.

Because many of these projects apply to disciplines outside the field of computer science, we do not expect that the average reader will already know about all of these "other" topics. Therefore, in our problem statements, we usually take considerable time to explain the topic as well as the problem. And we often explain how to go about solving the problem—in layman's terms. Therefore, working many of these projects will be like implementing computer solutions for customers who are not programmers themselves but understand their subject matter and know what they want you (the programmer) to do for them. They will explain their problem and how to go about solving it. But then they will expect you to create the program that actually solves that problem.

Because our project explanations frequently take considerable printed space, instead of putting them in the book itself, we put them on our website:

http://www.mhhe.com/dean3e

The following table provides a summary of the projects on the book's website. This table lists all of the book's projects in a sequence that matches the book's sequence. The first column identifies the first point at which you should be able to do the project, by chapter and section, in the form: ChapterNumber.SectionNumber. The second column is a unique project number for the chapter in question. The third column identifies the project's primary academic area with an abbreviation that's explained in the shorter table above. The fourth column indicates the difficulty relative to where you are in your study of Java. For example, what we call "easy" involves progressively more pages of code as you progress through the book. The last two columns provide a title and brief description of each project.

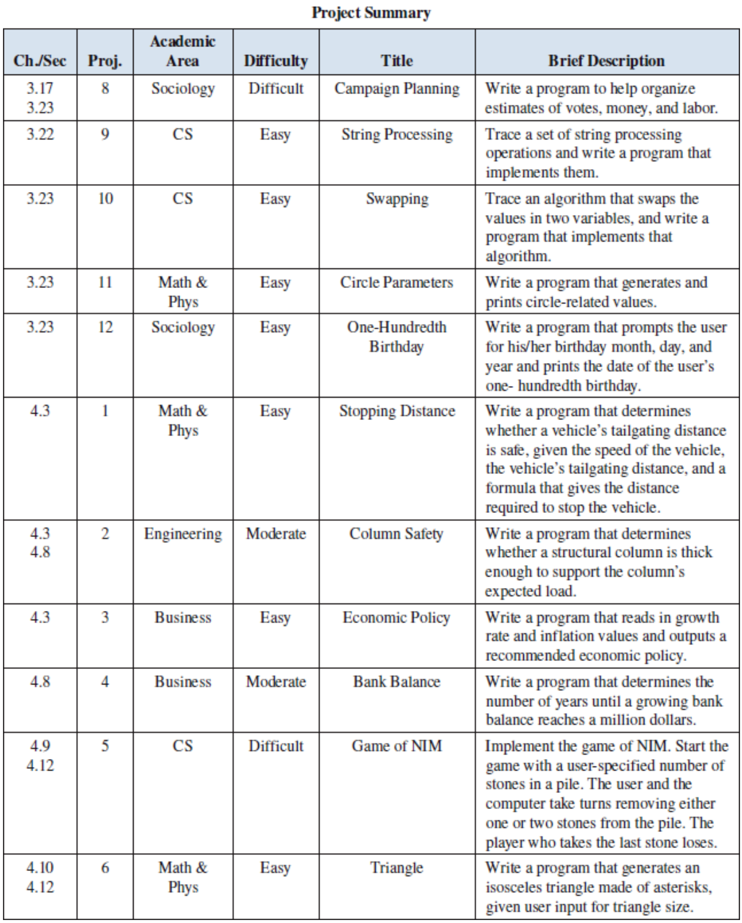


Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
2.7	1	Business	Easy	Annual Bonus–(Flowchart)	Draw a flowchart for an algorithm that computes an annual bonus.
2.7	2	Business	Easy	Annual Bonus—(Pseudocode)	Write pseudocode for an algorithm that computes an annual bonus.
2.7	3	Business	Easy	Number of Stamps—(Flowchart)	Draw a flowchart for an algorithm that calculates the number of stamps needed for an envelope. Use one stamp for every five sheets of paper.
2.7	4	Business	Easy	Number of Stamps—(Pseudocode)	Write pseudocode for an algorithm that calculates the number of stamps needed for an envelope. Use one stamp for every five sheets of paper.
2.7	5	Biol & Ecol	Moderate	Five Kingdoms—(Pseudocode)	Write pseudocode for an algorithm that identifies a biological kingdom from a set of characteristics.
2.7	6	Math & Phys	Easy	Speed of Sound—(Flowchart)	Draw a flowchart for an algorithm that provides the speed of sound in a particular medium.
2.7	7	Math & Phys	Easy	Speed of Sound—(Pseudocode)	Write pseudocode for an algorithm that provides the speed of sound in a particular medium.
2.7	8	Business	Moderate	Stock Market Return—(Flowchart)	Draw a flowchart for an algorithm that prints the type of market and its probability given a particular rate of return.
2.7	9	Business	Moderate	Stock Market Return—(Pseudocode)	Write pseudocode for an algorithm that prints the type of market and its probability given a particular rate of return.
2.8	10	Business	Moderate	Bank Balance—(Pseudocode)	Write pseudocode for an algorithm that determines the number of years until a growing bank balance reaches a million dollars.

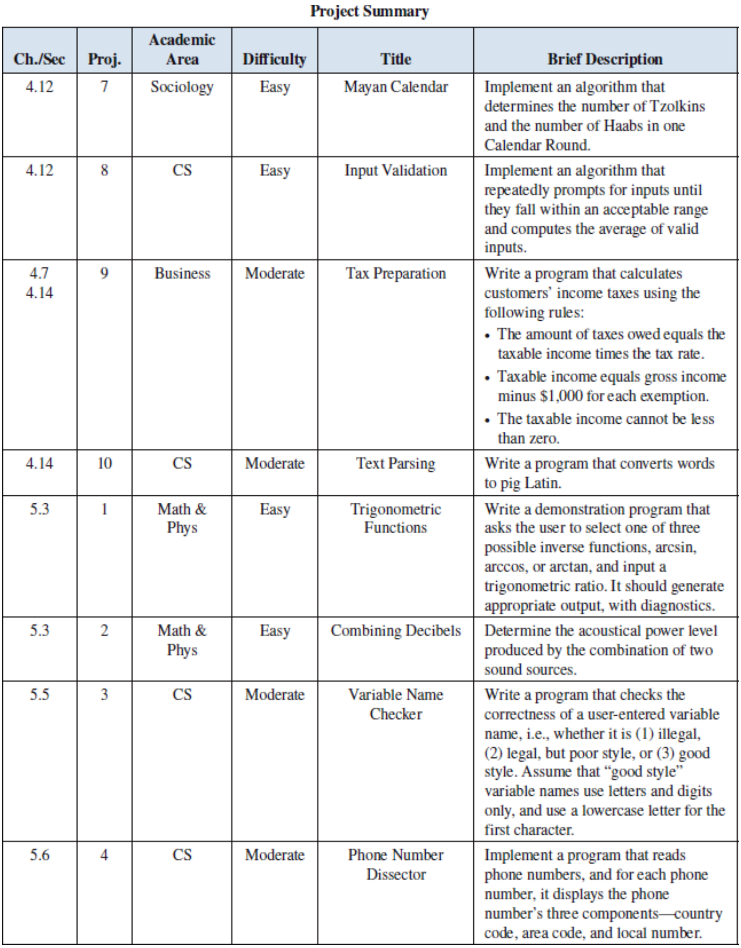
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
2.9	11	Engineering	Moderate	Loop Termination by User Query—(Flowchart)	Draw a flowchart for an algorithm that calculates the overall miles per gallon for a series of miles and gallons user inputs.
2.9	12	Engineering	Easy	Loop Termination by User Query—(Pseudocode)	Write pseudocode for an algorithm that calculates the overall miles per gallon for a series of miles and gallons user inputs.
2.9	13	Engineering	Moderate	Loop Termination by Sentinel Value—(Pseudocode)	Write pseudocode for an algorithm that calculates the overall miles per gallon for a series of miles and gallons user inputs.
2.9	14	Engineering	Easy	Loop Termination by Counter—(Pseudocode)	Write pseudocode for an algorithm that calculates the overall miles per gallon for a series of miles and gallons user inputs.
2.10	15	CS	Moderate	Average Weight—(Pseudocode)	Write pseudocode for an algorithm that determines average weight for a group of items.
3.2	1	CS	Easy	Hello World Experimentation	Experiment with the `Hello.java` program to learn the meanings of typical compile-time and runtime error messages.
3.3	2	CS	Moderate	Research	Study Oracle's Java Coding Conventions.
3.3	3	CS	Moderate	Research	Study Appendix 5, "Java Coding-Style Conventions."
3.16 3.23	4	Engineering	Difficult	Truss Analysis	Given the load in the center of a bridge and the weights of all truss members, compute the compression or tension force in each truss member.
3.17	5	CS	Easy	Sequence of Commands	Trace a sequence of commands and write a program that executes those commands.
3.17 3.23	6	CS	Moderate	Computer Speed	Given a simple set of hardware and software characteristics, write a program that estimates the total time to run a computer program.
3.17 3.23	7	Engineering	Moderate	HVAC Load	Calculate the heating and cooling loads for a typical residence.



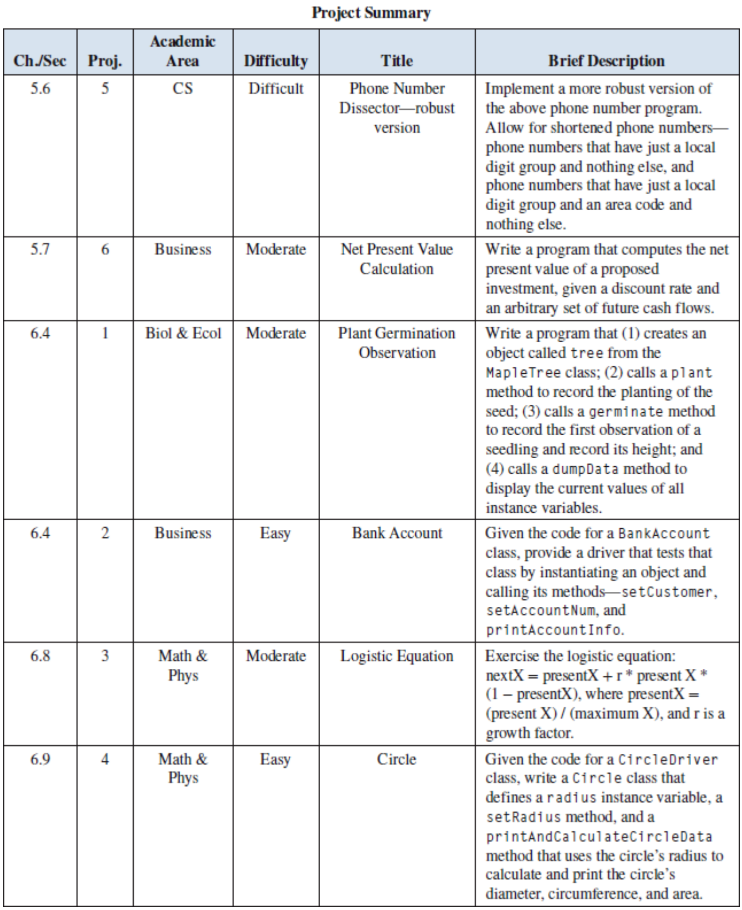
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
3.17 3.23	8	Sociology	Difficult	Campaign Planning	Write a program to help organize estimates of votes, money, and labor.
3.22	9	CS	Easy	String Processing	Trace a set of string processing operations and write a program that implements them.
3.23	10	CS	Easy	Swapping	Trace an algorithm that swaps the values in two variables, and write a program that implements that algorithm.
3.23	11	Math & Phys	Easy	Circle Parameters	Write a program that generates and prints circle-related values.
3.23	12	Sociology	Easy	One-Hundredth Birthday	Write a program that prompts the user for his/her birthday month, day, and year and prints the date of the user's one- hundredth birthday.
4.3	1	Math & Phys	Easy	Stopping Distance	Write a program that determines whether a vehicle's tailgating distance is safe, given the speed of the vehicle, the vehicle's tailgating distance, and a formula that gives the distance required to stop the vehicle.
4.3 4.8	2	Engineering	Moderate	Column Safety	Write a program that determines whether a structural column is thick enough to support the column's expected load.
4.3	3	Business	Easy	Economic Policy	Write a program that reads in growth rate and inflation values and outputs a recommended economic policy.
4.8	4	Business	Moderate	Bank Balance	Write a program that determines the number of years until a growing bank balance reaches a million dollars.
4.9 4.12	5	CS	Difficult	Game of NIM	Implement the game of NIM. Start the game with a user-specified number of stones in a pile. The user and the computer take turns removing either one or two stones from the pile. The player who takes the last stone loses.
4.10 4.12	6	Math & Phys	Easy	Triangle	Write a program that generates an isosceles triangle made of asterisks, given user input for triangle size.



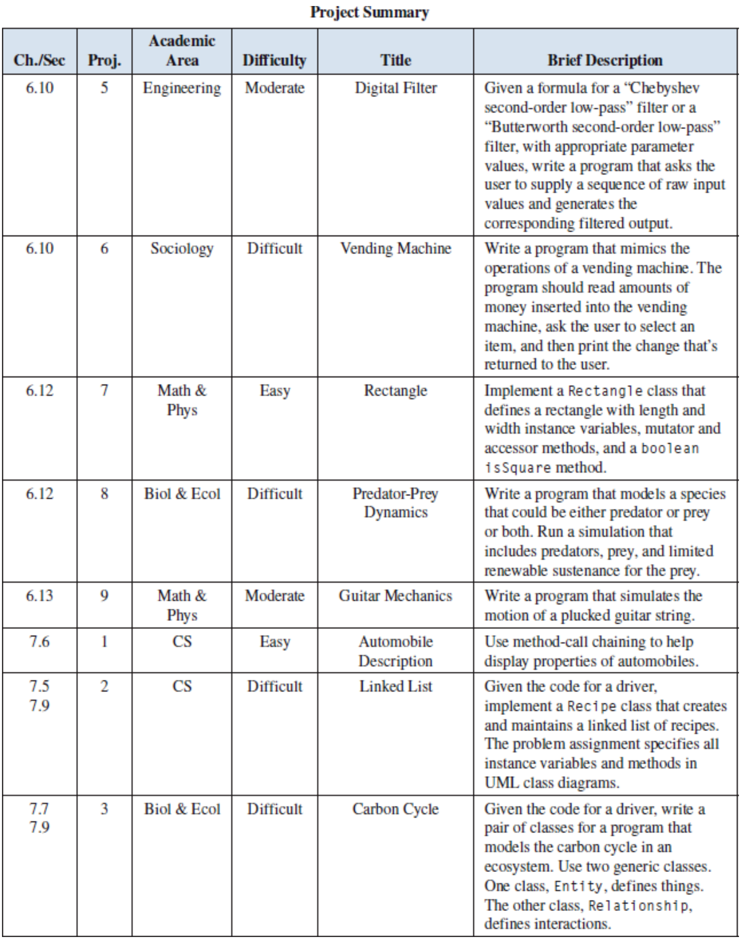
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
4.12	7	Sociology	Easy	Mayan Calendar	Implement an algorithm that determines the number of Tzolkins and the number of Haabs in one Calendar Round.
4.12	8	CS	Easy	Input Validation	Implement an algorithm that repeatedly prompts for inputs until they fall within an acceptable range and computes the average of valid inputs.
4.7 4.14	9	Business	Moderate	Tax Preparation	Write a program that calculates customers' income taxes using the following rules: • The amount of taxes owed equals the taxable income times the tax rate. • Taxable income equals gross income minus $1,000 for each exemption. • The taxable income cannot be less than zero.
4.14	10	CS	Moderate	Text Parsing	Write a program that converts words to pig Latin.
5.3	1	Math & Phys	Easy	Trigonometric Functions	Write a demonstration program that asks the user to select one of three possible inverse functions, arcsin, arccos, or arctan, and input a trigonometric ratio. It should generate appropriate output, with diagnostics.
5.3	2	Math & Phys	Easy	Combining Decibels	Determine the acoustical power level produced by the combination of two sound sources.
5.5	3	CS	Moderate	Variable Name Checker	Write a program that checks the correctness of a user-entered variable name, i.e., whether it is (1) illegal, (2) legal, but poor style, or (3) good style. Assume that "good style" variable names use letters and digits only, and use a lowercase letter for the first character.
5.6	4	CS	Moderate	Phone Number Dissector	Implement a program that reads phone numbers, and for each phone number, it displays the phone number's three components—country code, area code, and local number.



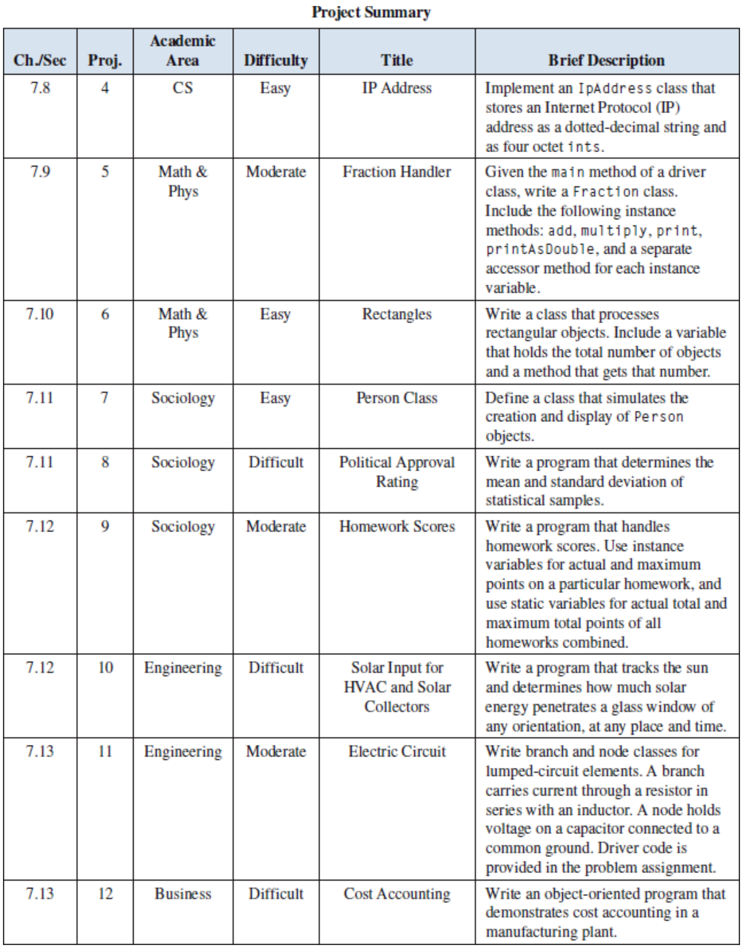
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
5.6	5	CS	Difficult	Phone Number Dissector—robust version	Implement a more robust version of the above phone number program. Allow for shortened phone numbers—phone numbers that have just a local digit group and nothing else, and phone numbers that have just a local digit group and an area code and nothing else.
5.7	6	Business	Moderate	Net Present Value Calculation	Write a program that computes the net present value of a proposed investment, given a discount rate and an arbitrary set of future cash flows.
6.4	1	Biol & Ecol	Moderate	Plant Germination Observation	Write a program that (1) creates an object called `tree` from the `MapleTree` class; (2) calls a `plant` method to record the planting of the seed; (3) calls a `germinate` method to record the first observation of a seedling and record its height; and (4) calls a `dumpData` method to display the current values of all instance variables.
6.4	2	Business	Easy	Bank Account	Given the code for a `BankAccount` class, provide a driver that tests that class by instantiating an object and calling its methods—`setCustomer`, `setAccountNum`, and `printAccountInfo`.
6.8	3	Math & Phys	Moderate	Logistic Equation	Exercise the logistic equation: nextX = presentX + r * present X * (1 – presentX), where presentX = (present X) / (maximum X), and r is a growth factor.
6.9	4	Math & Phys	Easy	Circle	Given the code for a `CircleDriver` class, write a `Circle` class that defines a `radius` instance variable, a `setRadius` method, and a `printAndCalculateCircleData` method that uses the circle's radius to calculate and print the circle's diameter, circumference, and area.



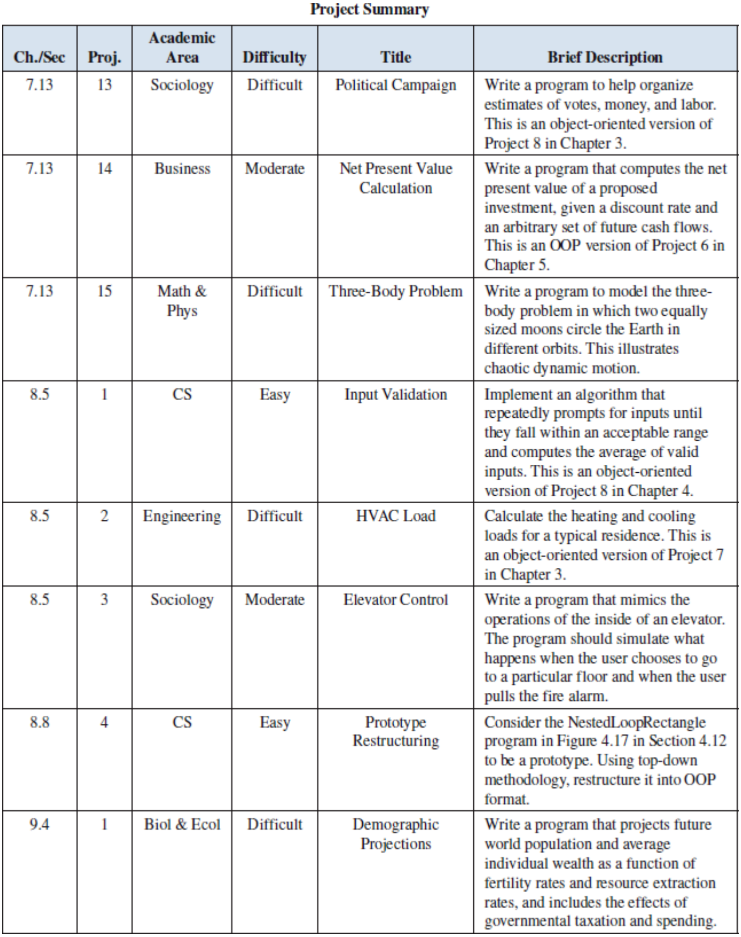
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
6.10	5	Engineering	Moderate	Digital Filter	Given a formula for a “Chebyshev second-order low-pass” filter or a “Butterworth second-order low-pass” filter, with appropriate parameter values, write a program that asks the user to supply a sequence of raw input values and generates the corresponding filtered output.
6.10	6	Sociology	Difficult	Vending Machine	Write a program that mimics the operations of a vending machine. The program should read amounts of money inserted into the vending machine, ask the user to select an item, and then print the change that’s returned to the user.
6.12	7	Math & Phys	Easy	Rectangle	Implement a `Rectangle` class that defines a rectangle with length and width instance variables, mutator and accessor methods, and a `boolean` `isSquare` method.
6.12	8	Biol & Ecol	Difficult	Predator-Prey Dynamics	Write a program that models a species that could be either predator or prey or both. Run a simulation that includes predators, prey, and limited renewable sustenance for the prey.
6.13	9	Math & Phys	Moderate	Guitar Mechanics	Write a program that simulates the motion of a plucked guitar string.
7.6	1	CS	Easy	Automobile Description	Use method-call chaining to help display properties of automobiles.
7.5 7.9	2	CS	Difficult	Linked List	Given the code for a driver, implement a `Recipe` class that creates and maintains a linked list of recipes. The problem assignment specifies all instance variables and methods in UML class diagrams.
7.7 7.9	3	Biol & Ecol	Difficult	Carbon Cycle	Given the code for a driver, write a pair of classes for a program that models the carbon cycle in an ecosystem. Use two generic classes. One class, `Entity`, defines things. The other class, `Relationship`, defines interactions.



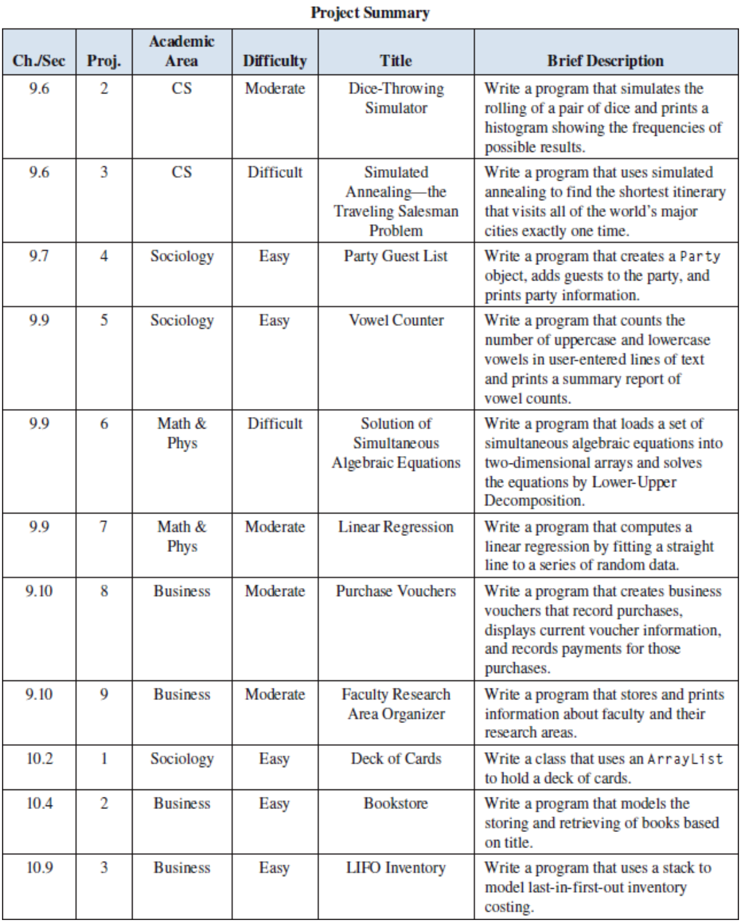
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
7.8	4	CS	Easy	IP Address	Implement an `IpAddress` class that stores an Internet Protocol (IP) address as a dotted-decimal string and as four octet `int`s.
7.9	5	Math & Phys	Moderate	Fraction Handler	Given the `main` method of a driver class, write a `Fraction` class. Include the following instance methods: `add`, `multiply`, `print`, `printAsDouble`, and a separate accessor method for each instance variable.
7.10	6	Math & Phys	Easy	Rectangles	Write a class that processes rectangular objects. Include a variable that holds the total number of objects and a method that gets that number.
7.11	7	Sociology	Easy	Person Class	Define a class that simulates the creation and display of `Person` objects.
7.11	8	Sociology	Difficult	Political Approval Rating	Write a program that determines the mean and standard deviation of statistical samples.
7.12	9	Sociology	Moderate	Homework Scores	Write a program that handles homework scores. Use instance variables for actual and maximum points on a particular homework, and use static variables for actual total and maximum total points of all homeworks combined.
7.12	10	Engineering	Difficult	Solar Input for HVAC and Solar Collectors	Write a program that tracks the sun and determines how much solar energy penetrates a glass window of any orientation, at any place and time.
7.13	11	Engineering	Moderate	Electric Circuit	Write branch and node classes for lumped-circuit elements. A branch carries current through a resistor in series with an inductor. A node holds voltage on a capacitor connected to a common ground. Driver code is provided in the problem assignment.
7.13	12	Business	Difficult	Cost Accounting	Write an object-oriented program that demonstrates cost accounting in a manufacturing plant.



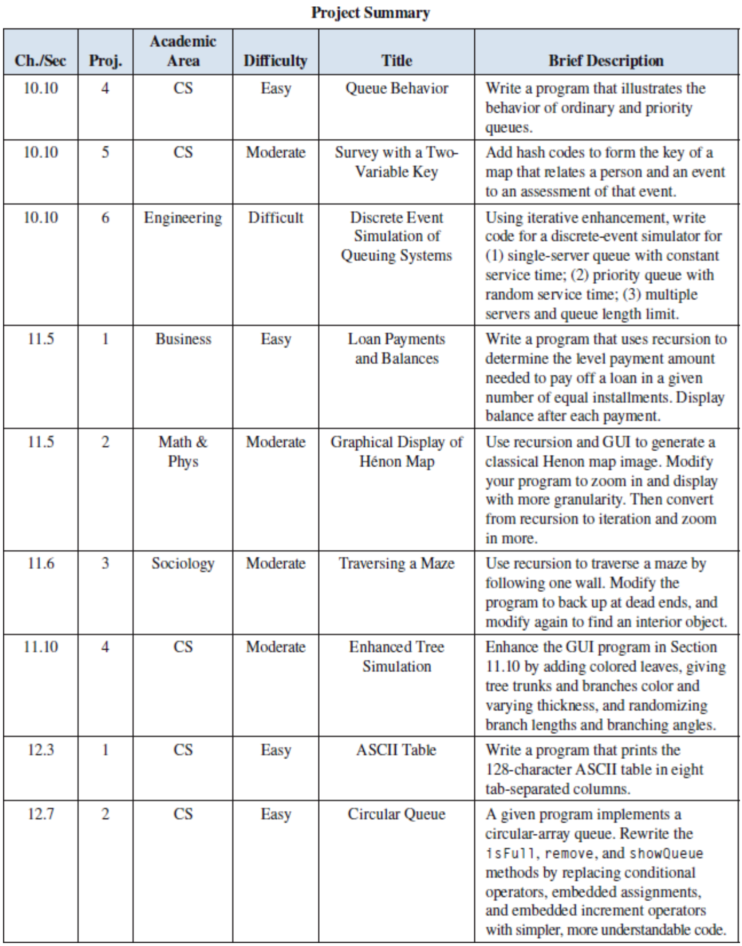
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
7.13	13	Sociology	Difficult	Political Campaign	Write a program to help organize estimates of votes, money, and labor. This is an object-oriented version of Project 8 in Chapter 3.
7.13	14	Business	Moderate	Net Present Value Calculation	Write a program that computes the net present value of a proposed investment, given a discount rate and an arbitrary set of future cash flows. This is an OOP version of Project 6 in Chapter 5.
7.13	15	Math & Phys	Difficult	Three-Body Problem	Write a program to model the three-body problem in which two equally sized moons circle the Earth in different orbits. This illustrates chaotic dynamic motion.
8.5	1	CS	Easy	Input Validation	Implement an algorithm that repeatedly prompts for inputs until they fall within an acceptable range and computes the average of valid inputs. This is an object-oriented version of Project 8 in Chapter 4.
8.5	2	Engineering	Difficult	HVAC Load	Calculate the heating and cooling loads for a typical residence. This is an object-oriented version of Project 7 in Chapter 3.
8.5	3	Sociology	Moderate	Elevator Control	Write a program that mimics the operations of the inside of an elevator. The program should simulate what happens when the user chooses to go to a particular floor and when the user pulls the fire alarm.
8.8	4	CS	Easy	Prototype Restructuring	Consider the NestedLoopRectangle program in Figure 4.17 in Section 4.12 to be a prototype. Using top-down methodology, restructure it into OOP format.
9.4	1	Biol & Ecol	Difficult	Demographic Projections	Write a program that projects future world population and average individual wealth as a function of fertility rates and resource extraction rates, and includes the effects of governmental taxation and spending.



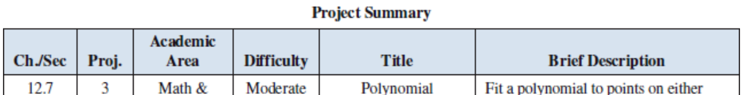
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
9.6	2	CS	Moderate	Dice-Throwing Simulator	Write a program that simulates the rolling of a pair of dice and prints a histogram showing the frequencies of possible results.
9.6	3	CS	Difficult	Simulated Annealing—the Traveling Salesman Problem	Write a program that uses simulated annealing to find the shortest itinerary that visits all of the world's major cities exactly one time.
9.7	4	Sociology	Easy	Party Guest List	Write a program that creates a `Party` object, adds guests to the party, and prints party information.
9.9	5	Sociology	Easy	Vowel Counter	Write a program that counts the number of uppercase and lowercase vowels in user-entered lines of text and prints a summary report of vowel counts.
9.9	6	Math & Phys	Difficult	Solution of Simultaneous Algebraic Equations	Write a program that loads a set of simultaneous algebraic equations into two-dimensional arrays and solves the equations by Lower-Upper Decomposition.
9.9	7	Math & Phys	Moderate	Linear Regression	Write a program that computes a linear regression by fitting a straight line to a series of random data.
9.10	8	Business	Moderate	Purchase Vouchers	Write a program that creates business vouchers that record purchases, displays current voucher information, and records payments for those purchases.
9.10	9	Business	Moderate	Faculty Research Area Organizer	Write a program that stores and prints information about faculty and their research areas.
10.2	1	Sociology	Easy	Deck of Cards	Write a class that uses an `ArrayList` to hold a deck of cards.
10.4	2	Business	Easy	Bookstore	Write a program that models the storing and retrieving of books based on title.
10.9	3	Business	Easy	LIFO Inventory	Write a program that uses a stack to model last-in-first-out inventory costing.



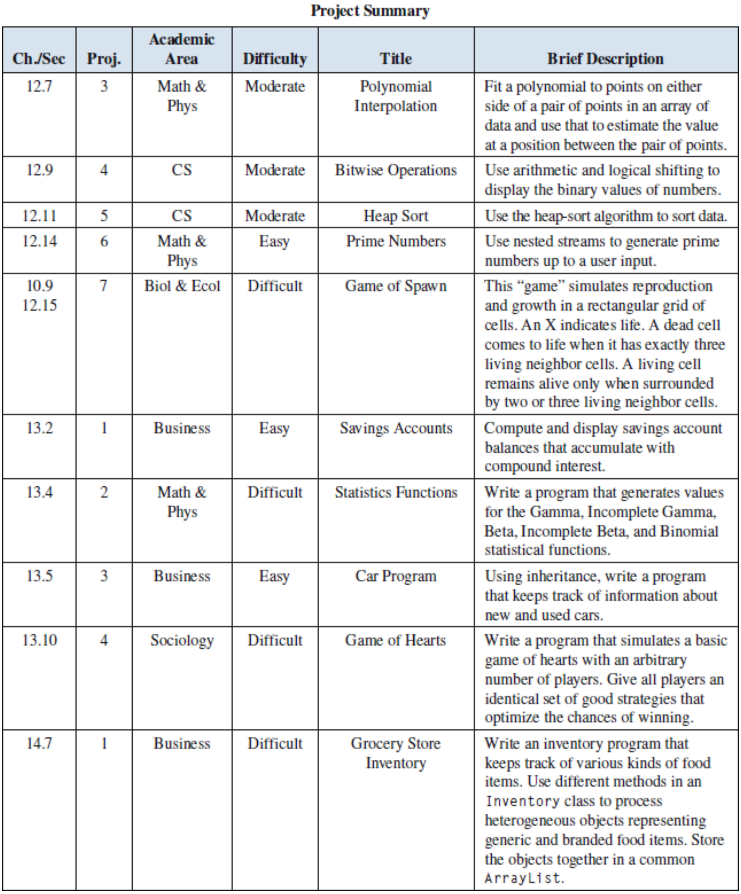
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
10.10	4	CS	Easy	Queue Behavior	Write a program that illustrates the behavior of ordinary and priority queues.
10.10	5	CS	Moderate	Survey with a Two-Variable Key	Add hash codes to form the key of a map that relates a person and an event to an assessment of that event.
10.10	6	Engineering	Difficult	Discrete Event Simulation of Queuing Systems	Using iterative enhancement, write code for a discrete-event simulator for (1) single-server queue with constant service time; (2) priority queue with random service time; (3) multiple servers and queue length limit.
11.5	1	Business	Easy	Loan Payments and Balances	Write a program that uses recursion to determine the level payment amount needed to pay off a loan in a given number of equal installments. Display balance after each payment.
11.5	2	Math & Phys	Moderate	Graphical Display of Hénon Map	Use recursion and GUI to generate a classical Henon map image. Modify your program to zoom in and display with more granularity. Then convert from recursion to iteration and zoom in more.
11.6	3	Sociology	Moderate	Traversing a Maze	Use recursion to traverse a maze by following one wall. Modify the program to back up at dead ends, and modify again to find an interior object.
11.10	4	CS	Moderate	Enhanced Tree Simulation	Enhance the GUI program in Section 11.10 by adding colored leaves, giving tree trunks and branches color and varying thickness, and randomizing branch lengths and branching angles.
12.3	1	CS	Easy	ASCII Table	Write a program that prints the 128-character ASCII table in eight tab-separated columns.
12.7	2	CS	Easy	Circular Queue	A given program implements a circular-array queue. Rewrite the `isFull`, `remove`, and `showQueue` methods by replacing conditional operators, embedded assignments, and embedded increment operators with simpler, more understandable code.



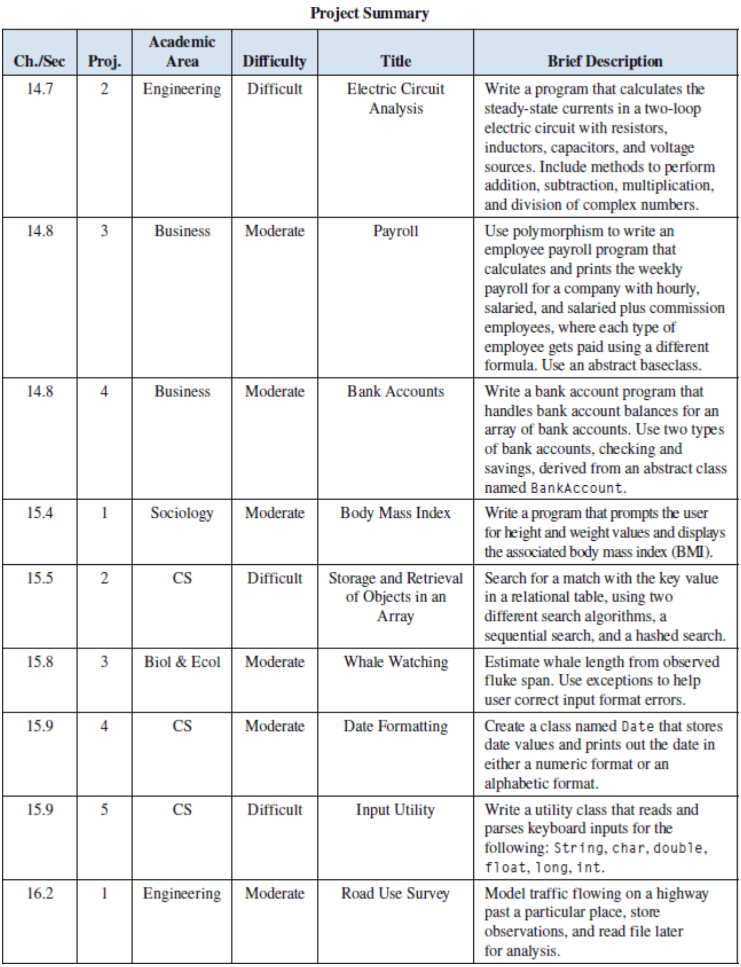
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
12.7	3	Math &	Moderate	Polynomial	Fit a polynomial to points on either



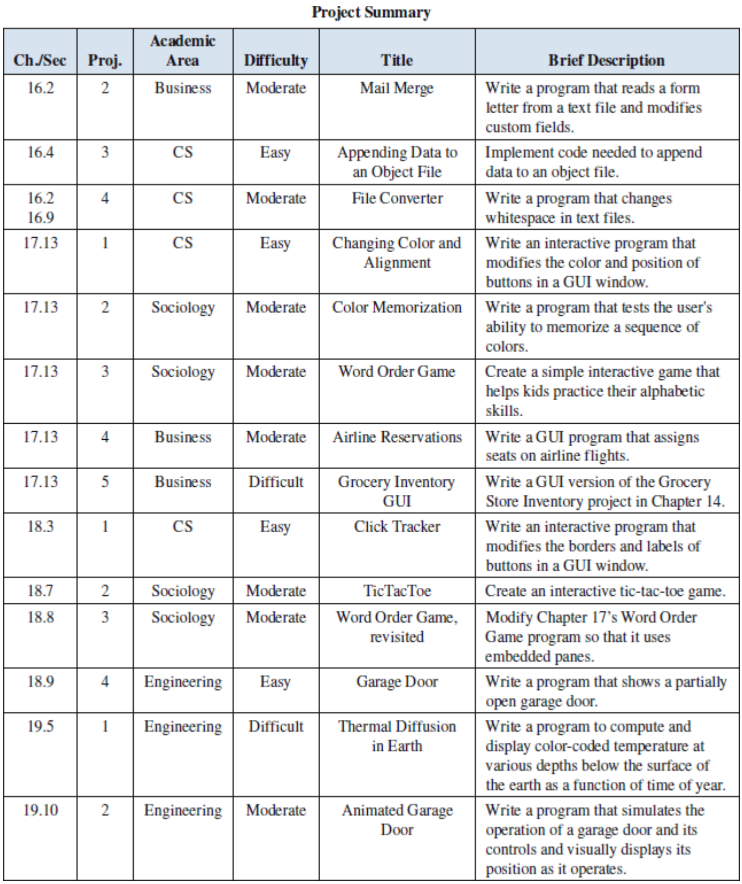
Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
12.7	3	Math & Phys	Moderate	Polynomial Interpolation	Fit a polynomial to points on either side of a pair of points in an array of data and use that to estimate the value at a position between the pair of points.
12.9	4	CS	Moderate	Bitwise Operations	Use arithmetic and logical shifting to display the binary values of numbers.
12.11	5	CS	Moderate	Heap Sort	Use the heap-sort algorithm to sort data.
12.14	6	Math & Phys	Easy	Prime Numbers	Use nested streams to generate prime numbers up to a user input.
10.9 12.15	7	Biol & Ecol	Difficult	Game of Spawn	This "game" simulates reproduction and growth in a rectangular grid of cells. An X indicates life. A dead cell comes to life when it has exactly three living neighbor cells. A living cell remains alive only when surrounded by two or three living neighbor cells.
13.2	1	Business	Easy	Savings Accounts	Compute and display savings account balances that accumulate with compound interest.
13.4	2	Math & Phys	Difficult	Statistics Functions	Write a program that generates values for the Gamma, Incomplete Gamma, Beta, Incomplete Beta, and Binomial statistical functions.
13.5	3	Business	Easy	Car Program	Using inheritance, write a program that keeps track of information about new and used cars.
13.10	4	Sociology	Difficult	Game of Hearts	Write a program that simulates a basic game of hearts with an arbitrary number of players. Give all players an identical set of good strategies that optimize the chances of winning.
14.7	1	Business	Difficult	Grocery Store Inventory	Write an inventory program that keeps track of various kinds of food items. Use different methods in an `Inventory` class to process heterogeneous objects representing generic and branded food items. Store the objects together in a common `ArrayList`.



Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
14.7	2	Engineering	Difficult	Electric Circuit Analysis	Write a program that calculates the steady-state currents in a two-loop electric circuit with resistors, inductors, capacitors, and voltage sources. Include methods to perform addition, subtraction, multiplication, and division of complex numbers.
14.8	3	Business	Moderate	Payroll	Use polymorphism to write an employee payroll program that calculates and prints the weekly payroll for a company with hourly, salaried, and salaried plus commission employees, where each type of employee gets paid using a different formula. Use an abstract baseclass.
14.8	4	Business	Moderate	Bank Accounts	Write a bank account program that handles bank account balances for an array of bank accounts. Use two types of bank accounts, checking and savings, derived from an abstract class named `BankAccount`.
15.4	1	Sociology	Moderate	Body Mass Index	Write a program that prompts the user for height and weight values and displays the associated body mass index (BMI).
15.5	2	CS	Difficult	Storage and Retrieval of Objects in an Array	Search for a match with the key value in a relational table, using two different search algorithms, a sequential search, and a hashed search.
15.8	3	Biol & Ecol	Moderate	Whale Watching	Estimate whale length from observed fluke span. Use exceptions to help user correct input format errors.
15.9	4	CS	Moderate	Date Formatting	Create a class named `Date` that stores date values and prints out the date in either a numeric format or an alphabetic format.
15.9	5	CS	Difficult	Input Utility	Write a utility class that reads and parses keyboard inputs for the following: `String`, `char`, `double`, `float`, `long`, `int`.
16.2	1	Engineering	Moderate	Road Use Survey	Model traffic flowing on a highway past a particular place, store observations, and read file later for analysis.



Project Summary

Ch./Sec	Proj.	Academic Area	Difficulty	Title	Brief Description
16.2	2	Business	Moderate	Mail Merge	Write a program that reads a form letter from a text file and modifies custom fields.
16.4	3	CS	Easy	Appending Data to an Object File	Implement code needed to append data to an object file.
16.2 16.9	4	CS	Moderate	File Converter	Write a program that changes whitespace in text files.
17.13	1	CS	Easy	Changing Color and Alignment	Write an interactive program that modifies the color and position of buttons in a GUI window.
17.13	2	Sociology	Moderate	Color Memorization	Write a program that tests the user's ability to memorize a sequence of colors.
17.13	3	Sociology	Moderate	Word Order Game	Create a simple interactive game that helps kids practice their alphabetic skills.
17.13	4	Business	Moderate	Airline Reservations	Write a GUI program that assigns seats on airline flights.
17.13	5	Business	Difficult	Grocery Inventory GUI	Write a GUI version of the Grocery Store Inventory project in Chapter 14.
18.3	1	CS	Easy	Click Tracker	Write an interactive program that modifies the borders and labels of buttons in a GUI window.
18.7	2	Sociology	Moderate	TicTacToe	Create an interactive tic-tac-toe game.
18.8	3	Sociology	Moderate	Word Order Game, revisited	Modify Chapter 17's Word Order Game program so that it uses embedded panes.
18.9	4	Engineering	Easy	Garage Door	Write a program that shows a partially open garage door.
19.5	1	Engineering	Difficult	Thermal Diffusion in Earth	Write a program to compute and display color-coded temperature at various depths below the surface of the earth as a function of time of year.
19.10	2	Engineering	Moderate	Animated Garage Door	Write a program that simulates the operation of a garage door and its controls and visually displays its position as it operates.

---

[1] Inevitably, we use a particular style for our pseudocode, but we repeatedly emphasize that other pseudocode styles are fine so long as they convey the intended meaning. Our pseudocode style is a combination of free-form description for high-level tasks and

more specific commands for low-level tasks. We've chosen a pseudocode style that is intuitive, to welcome new programmers, and structured, to accommodate program logic.

[2] We introduce GUI I/O at the end of Chapter 3 with dialog boxes. That opens up an optional door for GUI fans. If readers are so inclined, they can use the Alert, TextInputDialog, and ChoiceDialog classes to implement all our programs with GUI I/O rather than console I/O.

Introduction to Computers and Programming

Objectives

- Describe the various components that make up a computer.
- List the steps involved in program development.

- Know what it means to write algorithms using pseudocode.
- Know what it means to write programs with programming language code.
- Understand source code, object code, and the compilation process.
- Describe how bytecode makes Java portable.
- Become familiar with Java's history—why it was initially developed, how it got its name, and so forth.
- Enter, compile, and run a simple Java program.

Outline

1.1 Introduction 2

1.2 Hardware Terminology 2

1.3 Program Development 10

1.4 Source Code 12

1.5 Compiling Source Code into Object Code 13

1.6 Portability 14

1.7 Emergence of Java 15

1.8 Computer Ethics 18

1.9 First Program—Hello World 19

1.10 GUI Track: Hello World (Optional) 24

1.1 Introduction

This book is about problem solving. Specifically, it is about creating solutions to problems through a set of precisely stated instructions. We call such a set of instructions (when in a format that can be entered into and executed on a computer) a *program*. To understand what a program is, think about the following situation. Suppose you manage a department store, and you don’t know when to restock the shelves because you have difficulty keeping track of inventory. The solution to the problem is to write a set of instructions that keeps track of items as they arrive at your store and as they are purchased. If the instructions are correct and in a format that is understood by a computer, you can enter the instructions as a program, run the program, and enter item-arrival and item-purchase data as they occur. You can then retrieve inventory information from the computer any time you need it. That accurate and easily accessible knowledge enables you to restock your shelves effectively, and you are more likely to turn a profit.

The first step to learning how to write programs is to learn the background concepts. In this chapter, we teach background concepts. In subsequent chapters, we use these background concepts while explaining the really good stuff—how to program.

We start this chapter by describing the various parts of a computer. We then describe the steps involved in writing a program and in running a program. Next, we narrow our focus and describe the programming language we'll be using for the remainder of the book—Java. We present step-by-step instructions on how to enter and run a real Java program, so that you'll be able to gain some hands-on experience early on. We finish the chapter with an optional GUI-track section that describes how to enter and run a graphical user interface (GUI) program.

1.2 Hardware

Terminology

A *computer system* is made up of all the components that are necessary for a computer to operate and the connections between those components. There are two basic categories of components—*hardware* and *software*. Hardware consists of the physical components associated with a computer. Software consists of the programs that tell a computer what to do. For now, let's focus on hardware.

Our description of a computer's hardware provides you with the information you'll need as a beginning programmer. (A *programmer* is a person who writes programs.) After you master the material here, if you decide you want more, go to Webopedia's hardware web page at https://www.webopedia.com/Hardware.

The Big Picture

Figure 1.1 shows the basic hardware components in a computer system. It shows input devices at the left (keyboard, mouse, and scanner), output devices at the right (monitor and printer), storage devices at the bottom, and the central processing unit (CPU) and main memory in the center. The arrows in Figure 1.1 represent connections between the components. For example, the arrow from the keyboard to the CPU and main memory represents a cable (a connecting wire) that transmits information from the keyboard to the CPU and main memory. Throughout this section, we explain the CPU, main memory, and all the devices in Figure 1.1.

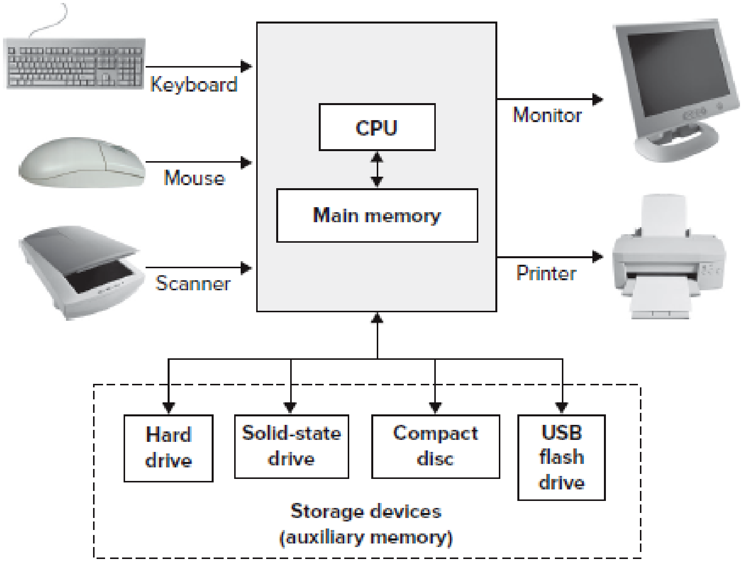

Keyboard
Mouse
Scanner
CPU
Main memory
Monitor
Printer
Hard drive
Solid-state drive
Compact disc
USB flash drive
Storage devices
(auxiliary memory)

Figure 1.1 A simplified view of a computer

Photodisc/Getty Images; Amos Morgan/Getty Images; Ryan McVay/Photodisc/Getty Images; Brand X Pictures/PunchStock/Getty Images; Ryan McVay/Photodisc/Getty Images.

Input and Output Devices

Input and output devices are collectively called *I/O devices.* There are different definitions of an *input device,* but usually the term refers to a device that transfers information into a computer. Remember—information going into a computer is input. For example, a keyboard is an input device because when a person presses a key, Page 3the keyboard sends information into the computer (it tells the computer which key was pressed).

There are different definitions of an *output device,* but usually the term refers to a device that transfers information out of a computer. Remember—information going out of a computer is output. For example, a *monitor* (also called a *display* or a *screen*) is an output device because it displays information going out of the computer.

Central Processing Unit

The *central processing unit,* often referred to as the *processor* or *microprocessor,* can be considered the computer's brain. As with a biological brain, the CPU splits its time between two basic activities—thinking and managing the rest of its system. The “thinking” activities occur when the CPU reads a program’s instructions and executes them. The “managing its system” activities occur when the CPU transfers information to and from the computer system’s other devices.

Here’s an example of a CPU’s thinking activities. Suppose you have a program that keeps track of a satellite’s position in its orbit around the Earth. Such a program contains quite a few mathematical calculations. The CPU performs those mathematical calculations.

Here’s an example of a CPU’s managing-its-system activities. Suppose you have a job application program. The program displays boxes in which a person enters his/her name, phone number, and so on. After entering information, the person uses his/her mouse and clicks a Done button. For such a program, the CPU manages its system as follows. To display the initial job application form, the CPU sends information to the monitor. To gather the person's data, the CPU reads information from the keyboard and mouse.

If you're thinking about buying a computer, you'll need to judge the quality of its components. To judge the quality of its components, you need to know certain component details. For CPUs, you should know the popular CPUs and the range of typical CPU speeds.

At the time of this third edition's writing:

- Popular CPUs—Core i9 (manufactured by Intel), Ryzen 9 (manufactured by AMD).
- Current CPU speeds—anywhere from 2.5 GHz up to 5.0 GHz.

What is *GHz,* you ask? GHz stands for *gigahertz* and is pronounced with hard *g*'s, as in *giggle. Giga* means billion, and *hertz* is a unit of measure that deals with the number of times that something occurs per second. A 2.5 GHz CPU uses a clock that ticks 2.5 billion times per second. That's fast, but a 5.0 GHz CPU is even faster—it uses a clock that ticks 5.0 billion times per second. A CPU's clock speed provides a rough measure for how fast the CPU gets things done. Clock ticks are the initiators for computer tasks. With more clock ticks per second, there are more opportunities for getting tasks done. Another way to get more things done in a given time is to do them in parallel. An Intel i9 has eight cores and can keep track of sixteen threads. In other words, this CPU chip contains within itself eight independent processing units and it can keep track of up to sixteen sequential operations.

Main Memory

When a computer executes instructions, it often needs to save intermediate results. For example, in calculating the average speed for 100 speed measurements, the CPU needs to calculate the sum of all the speed values prior to dividing by the number of measurements. The CPU calculates the sum by creating a storage area for it. For each speed value, the CPU adds the value to the sum storage area. Think of memory as a collection of storage boxes. The sum is stored in one of memory's storage boxes.

There are two categories of memory—*main memory* and *auxiliary memory*. The CPU works more closely with main memory. Think of main memory as a storage room next to the boss's office. The boss is the CPU, and he/she stores things in the storage room's storage boxes whenever the need arises. Think of auxiliary memory as a warehouse that's across the street from the boss's building. The boss uses the warehouse to store things, but doesn't go there all that often. Since auxiliary memory is considered secondary to main memory, auxiliary memory is sometimes referred to as *secondary storage*. We'll consider auxiliary memory details in the next subsection. For now, we'll focus on main memory details.

The CPU relies on main memory a lot. It's constantly storing data in main memory and reading data from main memory. With this constant interaction, it's important that the CPU and main memory are able to communicate quickly. To ensure quick communication, the CPU and main memory are physically close together. They are both constructed on *chips,* and they both plug into the computer's main circuit board, the *motherboard.*

See Figure 1.2 for a picture of a motherboard, a CPU chip, and main memory chips.

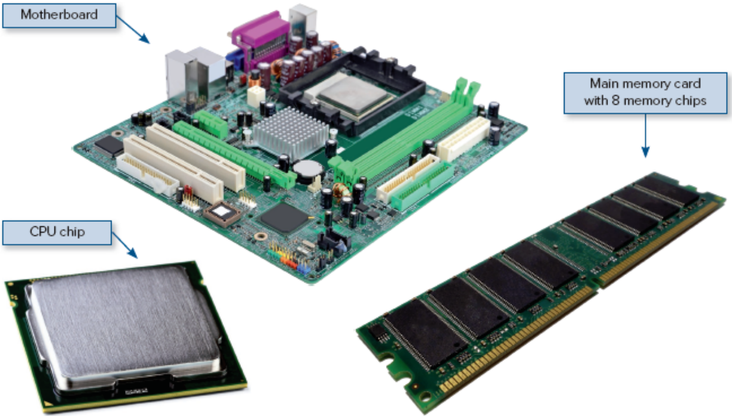


Figure 1.2 Motherboard, CPU chip, and main memory chips

Hellen Sergeyeva/123RF; Oleksandr Chub/Shutterstock; tonytao/Shutterstock.

Main memory contains storage boxes, and each storage box contains a piece of information. For example, if a program stores our last name, Dean, it uses eight storage boxes: one for the first half of D, one for the second half of D, one for the first half of e, one for the second half of e, and so on. After storing the four letters, the program will probably need to retrieve them at some point later on. For information to be retrievable, it must have an address. An *address* is a specifiable location. A postal address uses street, city, and ZIP code values to specify a location. A computer address uses the information's position within main memory to specify a location. Main memory's first storage box is at the zero position, so we say it's at address 0. The second storage box is at the one position, so we say it's at address 1. See Figure 1.3. It shows Dean stored in memory starting at address 50,000.

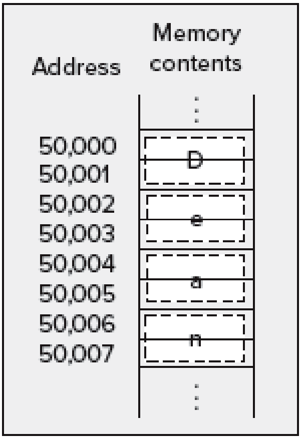

Address
Memory contents
50,000
50,001
50,002
50,003
50,004
50,005
50,006
50,007
D
e
a
n

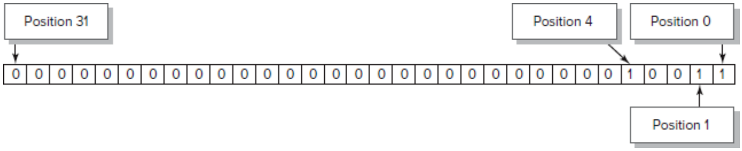
Figure 1.3 The characters D, e, a, n stored in memory starting at address 50,000

It’s important to understand the formal terminology when talking about the size of main memory. Suppose you’re buying a computer and you want to know how big the computer’s main memory is. If you ask a salesperson how many “storage boxes” it contains, you’ll probably get a perplexed look. What you need to do is ask about its *capacity*—that’s the formal term for its size. If you ask for the main memory’s capacity, the salesperson will say something like, “It’s 1 *gigabyte.*” You already know that *giga* means billion. A *byte* refers to the size of one storage box. So a 1 gigabyte capacity main memory holds 1 billion storage boxes.

Let’s describe storage boxes in more detail. You know that storage boxes can hold characters, like the letter D. But computers aren’t very smart—they don’t understand the alphabet. They only understand 0’s and 1’s. So computers map each alphabet character to a series of sixteen 0’s and 1’s. For example, the letter D is 00000000 01000100. So in storing the letter D, main memory actually stores 00000000 01000100. Each of the 0’s and 1’s is called a *bit.* And each of the eight-bit groupings is called a *byte*.

Are you wondering why computers use 0’s and 1’s? Computers understand only high-energy signals versus low-energy signals. When a computer generates a low-energy signal, that’s a 0. When a computer generates a high-energy signal, that’s a 1.

You know that computers store characters as 0’s and 1’s, but did you know that computers also store numbers as 0’s and 1’s? Formally, we say that computers use the *binary number system.* The binary number system uses just two digits, 0 and 1, to represent all numbers. For example, computers store the number 19 as 32 bits, 00000000 00000000 00000000 00010011. The reason those 32 bits represent 19 is that each 1-value bit represents a power of 2. Note that there are three 1-value bits. As shown here, the 1-value bits are at positions 0, 1, and 4, where the positions start at 0 from the right side. A bit's position determines its power of 2. Thus, the rightmost bit, at position 0, represents 2 raised to the power 0, which is 1 ($2^0 = 1$). The bit at position 1 represents 2 raised to the power 1, which is 2 ($2^1 = 2$). And the bit at position 4 represents 2 raised to the power 4, which is 16 ($2^4 = 16$). Add the three powers and you get 19 ($1 + 2 + 16 = 19$). Voila! Appendix 8 contains additional information about the binary number system. Feel free to peruse the appendix now, or wait until after you've read Chapter 12, which introduces another number system—hexadecimal. The appendix provides an in-depth discussion of the binary and hexadecimal number systems, as well as a third number system—octal.


Position 31
Position 4
Position 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 1
Position 1

Be aware that main memory is often referred to as *RAM*. RAM stands for *random access memory*. Main memory is considered “random access” because data can be directly accessed at any address; that is, at a “random” address. That’s in contrast to some storage devices where data is accessed by starting at the very beginning and stepping through all the data until the target data is reached.

Once again, if you’re buying a computer, you’ll need to judge the quality of its components. For the main memory/RAM component, you’ll need to know whether its capacity is adequate. At the time of this book's writing typical main memory capacities range from 4 GB up to 16 GB, where *GB* stands for *gigabyte,* but you'll see refurbished computers with significantly greater main memory capacity.

Auxiliary Memory

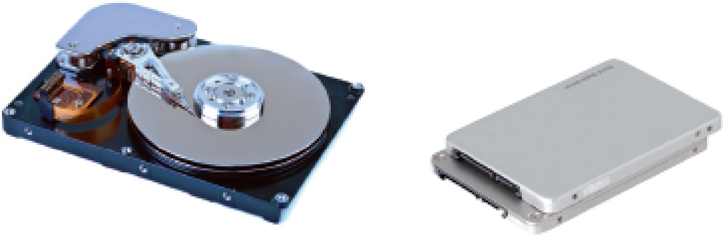
Main memory is *volatile,* which means that data is lost when power to the computer goes off. You might ask: If data is lost when power goes off, how can anyone save anything permanenly on a computer? The answer is something you do (or should do) frequently. When you perform a save command, the computer makes a copy of the main memory data you're working on and stores the copy in auxiliary memory. Auxiliary memory is *nonvolatile,* which means that data is not lost when power to the computer goes off.

One advantage of auxiliary memory over main memory is that it’s nonvolatile. Another advantage is that its cost per unit of storage is much less than main memory’s cost per unit of storage. A third advantage is that it is more *portable* than main memory (i.e., it can be moved from one computer to another more easily).

The disadvantage of auxiliary memory is that its *access time* is quite a bit slower than main memory’s access time. Access time is the time it takes to locate a single piece of data and make it available to the computer for processing.

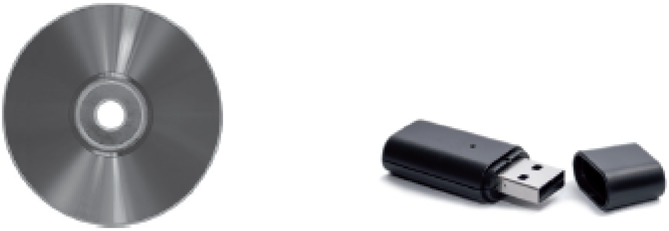
Auxiliary memory comes in many different forms, the most common of which are hard drives, solid-state drives (SSDs), universal serial bus (USB) flash drives, and compact discs. All these devices are called *storage media,* or simply *storage devices*. Figure

1.4 shows pictures of them.



Hard drive

Solid-state drive



Compact disc

USB flash drive

Figure 1.4 Hard drive, solid-state drive, compact disc, and USB flash drive

PaulPaladin/Shutterstock; Howard Kingsnorth/Photodisc/Getty Images; Scanrail/Dreamstime.com; MMXeon/ Shutterstock; chanchai howharn/ Shutterstock.

Hard Drives and Solid-State Drives

Hard drives and *solid-state drives* serve the same basic purpose: They provide the primary permanent storage for a computer. They have different advantages and disadvantages, which make them attractive to different types of computers. Most *desktop* computers (computers that remain stationary on or next to a desk) use hard drives. On the other hand, many *laptop* computers (computers that are portable enough to sit on someone's lap) and almost all *tablet* computers (computers that use a touch screen for their primary input device, rather than a keyboard and mouse) use solid-state drives instead of hard drives.

Solid-state drives are particularly suitable for laptop and tablet computers because they have no moving mechanical parts. As such, they are more resistant to damage when subject to travel. Also, they are smaller and lighter than hard drives, which once again makes them particularly suitable for laptop and tablet computers. A disadvantage of solid-state drives is their cost: Given a solid-state drive and a hard drive with the same capacity, the solid-state drive will be quite a bit more expensive. Thus, solid-state drives can be cost prohibitive for computers with large capacity needs. With their large capacity needs, most desktop computers still use hard drives (although solid-state drives are becoming more popular as prices fall). The trend is for portable computers to rely on *cloud storage,* where computers transmit their data over the Internet for storage on pools of computers hosted by third-party data centers. With cloud storage, portable computers' local storage needs are reduced and solid-state drives become more affordable.

Hard drives are slower than SSDs because for a computer to access a particular piece of data on a hard drive's disk, the computer must wait for the disk to spin the data to the place where the data can be read. The spinning and reading mechanisms constitute the hard drive's drive. As the drive rotates its disk (or disks), its *heads* (electronic sensors) access the disk's data as it spins past. Be aware that a hard drive is sometimes referred to as a *hard disk drive* (HDD).

Access to solid-state-drive storage is faster than access to hard drive storage because solid-state drives don't have to wait for mechanical parts to move. Instead, they simply have to wait for - electronic signals to arrive. Electronic signals, traveling at the speed of light, move much faster than spinning disks.

Off-line Storage

Although hard drives and solid-state drives are sometimes located outside a computer's metal case and connected to the computer with a cable (such drives are called *external drives*), in the interest of speed, most hard drives and solid-state drives are located inside a computer's metal case. Their internal location makes it difficult for them to be transferred from one computer to another. On the other hand, *off-line storage devices,* such as USB flash drives and compact discs, can be easily transferred from one computer to another because they are designed to connect and disconnect to and from a computer easily.

USB flash drives, also called *thumb drives,* are particularly portable because they are the size of a person's thumb and they can be *hot swapped* into virtually any computer. (Hot swapping is when you plug a device into a computer while the computer is on.) The "USB" in USB flash drive stands for "universal serial bus," and it refers to a particular type of connection to the computer. More specifically, it refers to a particular type of connection wire and connection socket. A flash drive uses that type of connection, and we therefore call it a *USB flash drive.* USB flash drives plug into USB ports, where *port* is the formal term for a connection socket. USB ports are ubiquitous on computers, and that is another reason that USB flash drives are particularly portable.

USB flash drives are built with *flash memory,* which is a popular form of nonvolatile storage with no moving mechanical parts. Solid-state drives also use flash memory. However, USB flash drives are much slower than solid-state drives (and slightly slower than hard drives) because USB flash drives are connected to the computer with a narrow, relatively slow USB interface, whereas solid-state drives are connected to the rest of the computer with a wide, relatively fast interface.

Compact discs provide a less expensive and slower form of off-line storage. The most popular types of compact discs can be grouped as follows:

- CD-Audio—for storing recorded music, usually referred to as just "CD" (for compact disc)
- CD-ROM, CD-R, CD-RW—for storing computer data and recorded music
- DVD, DVD-R, DVD-RW—for storing video, computer data, and recorded music
- Blu-ray—for storing video and computer data, designed to replace DVDs

The "ROM" in CD-ROM stands for "read-only memory." *Read-only* memory refers to memory that can be read from, but not written to. Thus, you can read a CD-ROM, but you can't change its contents. With CD-Rs, you can write once and read as many times as you like. With CD-RWs, you can write and read as often as you like.

DVD stands for “digital versatile disc” or “digital video disc.” DVDs parallel CD-ROMs in that you can read from them, but you can’t write to them. Likewise, DVD-Rs and DVD-RWs parallel CD-Rs and CD-RWs in terms of their reading and writing capabilities.

Blu-ray, also known as Blu-ray Disc (BD), is the name of an optical disc format that was invented to store high-definition videos and large amounts of data. The technology is called Blu-ray because unlike DVDs, which use a red laser, Blu-ray discs are accessed using a blue-violet laser. It was originally thought that Blu-ray discs would replace DVDs fairly quickly, but it still hasn’t happened because (1) Blu-ray is more expensive (the discs and the drives), (2) Blu-ray hardware is backward compatible (so DVDs can run on Blu-ray players), and (3) streaming is becoming more popular, so auxiliary CDs, DVDs, and Blu-ray are all taking a hit.

Storage Capacity Comparison

Different storage devices have different storage capacities. At the time of this book's writing:

- Typical hard drives have a capacity range from 450 GB up to 4 TB (*TB* stands for *terabyte,* where *tera* is 1 trillion).
- Typical solid-state drives have a capacity range from 250 GB up to 1 TB.
- Typical USB flash drives have a capacity range from 8 GB up to 1 TB.
- Typical CD-ROMs, CD-Rs, and CD-RWs have a capacity of 700 MB (*MB* stands for megabyte,

where *mega* is 1 million).

- Typical DVDs, DVD-Rs, and DVD-RWs have a capacity range from 4.7 GB up to 9.4 GB.
- Typical Blu-ray discs have a capacity range from 25 GB up to 100 GB.

File Access

To access data on your computer, you'll need to access the file that contains the data. A *file* is a group of related instructions or a group of related data. For example, (1) a program is a file that holds a set of instructions, and (2) a Word document is a file that holds text data created by Microsoft Word.

Files are stored on auxiliary memory storage devices. In order to retrieve a file (for the purpose of viewing, copying, etc.), you need to specify the storage device on which the file is stored. On computers that use Microsoft Windows, the different storage devices are specified using a drive letter followed by a colon. If your computer has a hard drive or a solid-state drive, your computer will refer to one of the drives using drive letter C (C:). If your computer has additional hard drives or solid-state drives, it will refer to them using subsequent drive letters (D:, E:, etc.). If your computer has compact-disc drives, it will refer to them using the first unused drive letters starting no earlier than D:. If your computer has additional storage devices, such as external hard drives and USB flash drives, it will refer to them using the next unused drive letters, starting no earlier than D:.

You might have noticed that drive letters A and B were not mentioned in this discussion so far. In the past, A: and B: were used for floppy disk drives. *Floppy disks* (also called *diskettes*) are off-line storage devices that were very popular from the mid-1970s through about 2005. They are called “floppy” because the original forms of these devices would bend, in contrast to the disks in hard drives, which do not bend. Computer manufacturers no longer provide floppy disk drives because floppy disks have been superseded by more durable, greater capacity off-line storage devices, such as USB flash drives.

Even though floppy disks are no longer used, their legacy lives on. Because Windows-based computers reserved drive letters A and B for floppy disk drives in the past, Windows-based computers continue to start with drive letter C for hard drives and solid-state drives. Because floppy disks became synonymous with file storage in the 1980s and 1990s, software manufacturers introduced floppy disk icons that, when clicked, would save the user's current file. Using a floppy disk icon for a file-save operation is still the norm today. This standard floppy disk icon should look familiar: 💾

Common Computer-Hardware Vocabulary

When buying a computer or when talking about computers with your computer friends, you'll want to make sure to understand the vernacular—the terms that people use in everyday speech as opposed to the terms found in textbooks—so that you will be able to understand what's going on. When a computer-savvy person refers to a computer's memory by itself, the person typically means main memory—the computer's RAM. When someone refers to a computer's *disk space,* the person typically means the capacity of the computer's hard drive.

Pace of Computer Improvements

For as long as memory and CPU components have been around, manufacturers of these devices have been able to improve their products' performances at a consistently high rate. For example, RAM and hard drive capacities double approximately every two years. CPU speeds also double approximately every two years.

An *urban legend* is a story that spreads spontaneously in various forms and is popularly believed to be true. The following exchange is a classic Internet urban legend that comments on the rapid pace of computer improvements.[1] Although the exchange never took place, the comments, particularly the first one, are relevant.

At a COMDEX computer exposition in the 1990s, Bill Gates reportedly compared the computer industry with the auto industry and stated, “If GM had kept up with the technology like the computer industry has, we would all be driving $25.00 cars that got 1,000 miles to the gallon.”

In response to Bill’s comments, General Motors issued a press release stating:

If GM had developed technology like Microsoft, we would all be driving cars with the following characteristics:

1. For no reason whatsoever, your car would crash twice a day.

2. Every time they repainted the lines in the road, you would have to buy a new car.

3. Occasionally your car would die on the freeway for no reason. You would just accept this, restart, and drive on.

4. Apple would make a car powered by the sun, reliable, five times as fast, and twice as easy to drive—but it would run on only 5 percent of the roads.

5. The oil, water temperature, and alternator warning lights would be replaced by a single “This Car Has Performed an Illegal Operation” warning light.

6. Occasionally, for no reason whatsoever, your car would lock you out and refuse to let you in until you simultaneously lifted the door handle, turned the key, and grabbed the radio antenna.

7. The airbag system would ask “Are you sure?” before deploying.

1.3 Program Development

As mentioned earlier, a program is a set of instructions that can be used to solve a problem. Often, a program contains many instructions, and the instructions are rather complicated.

Therefore, developing a successful program requires some effort. It requires careful planning, careful implementation, and ongoing maintenance. Here is a list of typical steps involved in the program development process:

- Requirements analysis
- Design
- Implementation
- Testing
- Documentation
- Maintenance

Page 11

Requirements analysis is determining the program's needs and goals. *Design* is writing a rough outline of the program. *Implementation* is writing the program itself. *Testing* is verifying that the program works. *Documentation* is writing a description of what the program does. *Maintenance* is making improvements and fixing errors later on. The steps are ordered in a reasonable sequence in that you'll normally perform requirements analysis first, design second, and so on. But some of the steps should be performed throughout the development process rather than at one particular time. For example, you should work on the documentation step throughout the development process, and you should work on the testing step during and after the implementation step and also after the maintenance step. Be aware that you'll often need to repeat the sequence of steps as needs arise. For example, if one of the program's goals changes, you'll need to repeat all of the steps in varying degrees.

We discuss the requirements analysis step and the design step in this section. We discuss the design step in detail in Chapter 2, and we illustrate it with examples throughout the book. We discuss the implementation step in this chapter's "Source Code" section, and we illustrate it with examples throughout the book. We discuss the testing step in Chapter 8. We discuss the documentation step starting in Chapter 3 and illustrate it with examples throughout the book. We discuss the maintenance step in Chapter 8 and illustrate it with examples throughout the book.

Requirements Analysis

The first step in the program development process is a requirements analysis, where you determine the needs and goals of your program. It's important that the programmer thoroughly understands the customer's wishes. Unfortunately, it's all too common for a programmer to produce a program, only to find out later that the customer wanted something different. This unfortunate circumstance can often be blamed on imprecise communication between the customer and the programmer at the beginning of the project. If a customer and programmer rely solely on a verbal description of the proposed solution, it’s easy to omit important details. Later on, those omitted details can be a problem when the customer and programmer realize that they had different assumptions about how the details would be implemented.

To aid the up-front communication process, the customer and programmer should create *screen shots* of data-entry screens and output reports. A screen shot is a picture of what the computer screen looks like. To create screen shots, you can write short programs that print data-entry screens with hypothetical input, and you can write short programs that print reports with hypothetical results. As a quicker alternative, you can create screen shots with the help of drawing software or, if you’re a decent artist, with pencil and paper.

Program Design

After the requirements analysis step, the second step is program design, where you write a draft of your program and focus on the basic logic, not the wording details. More specifically, you write instructions that are coherent and logically correct, but you don’t worry about missing minor steps or misspelling words. That sort of program is referred to as an *algorithm*. For example, a cake recipe is an algorithm. It contains instructions for solving the problem of baking a cake. The instructions are coherent and logically correct, but they don’t contain every minor step, like covering your hands with pot holders prior to removing the cake from the oven.

Pseudocode

In writing an algorithm, you should focus on organizing the flow of the instructions, and you should try to avoid getting bogged down in details. To facilitate that focus, programmers often write an algorithm's instructions using *pseudocode*. Pseudocode is an informal language that uses regular English terms to describe a program's steps. With pseudocode, precise computer *syntax* is not required. Syntax refers to the words, grammar, and punctuation that make up a language. Pseudocode syntax is more forgiving: Pseudocode must be clear enough so that humans can understand it, but the words, grammar, and punctuation don't have to be perfect. We mention this leniency in order to contrast it with the precision required for the next phase in a program's development. In the next section, we'll cover the next phase, and you'll see that it requires perfect words, grammar, and punctuation.

Example—Using Pseudocode to Find Average Miles per Hour

Suppose you are asked to write an algorithm that finds the average miles per hour value for a given car trip. Let’s step through the solution for this problem. To determine the average miles per hour, you’ll need to divide the total distance traveled by the total time. Let’s assume that you have to calculate the total distance from two given locations. To determine the total distance, you’ll need to take the ending-point location, called “ending location,” and subtract the starting-point location, called “starting location,” from it. Let’s assume that you have to calculate the total time in the same manner, subtracting the starting time from the ending time.

Putting it all together, the pseudocode for calculating average miles per hour looks like this:

```
Calculate ending location minus starting location.
Put the result in total distance.
Calculate ending time minus starting time.
Put the result in total time.
Divide total distance by total time.
```

At this point, some readers might want to learn about a relatively advanced form of program development—object-oriented programming, or OOP as it's commonly called. OOP is the idea that when you're designing a program, you should first think about the program's components (objects) rather than the program's tasks. You don't need to learn about OOP just yet, and you're not properly prepared to learn about

OOP implementation details, but if you're interested in a high-level overview, you can find it in Chapter 6, Section 6.2.

1.4 Source Code

In the early stages of a program's development, you write an algorithm using pseudocode. Later, you translate the pseudocode to *source code*. Source code is a set of instructions written in a programming language.

Programming Languages

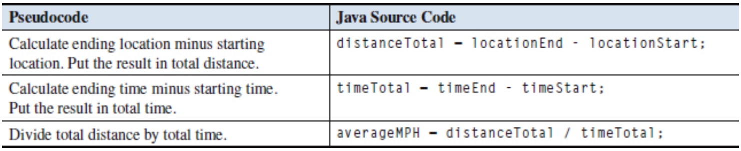
A *programming language* is a language that uses specially defined words, grammar, and punctuation that a computer understands. If you try to run pseudocode instructions on a computer, the computer won't understand them. On the other hand, if you try to run programming language instructions (i.e., source code) on a computer, the computer will understand them.

Just as there are many spoken languages in the world (English, Chinese, Hindi, etc.), there are many programming languages as well. Some of the more popular programming languages are Java, C++, and Python. Each programming language has its own set of syntax rules. In this book, we'll focus on the Java programming language. If you write your program in Java, you must follow Java's syntax rules precisely in terms of words, grammar, and punctuation. If you write Java source code using incorrect syntax (e.g., you misspell a word or forget necessary punctuation, like a semicolon), and you try to run such source code on a computer, the computer won't be able to understand it.

Example—Using Java to Find Average Miles per Hour

Continuing with the earlier example where you wrote pseudocode to find the average miles per hour value for a given car trip, let’s now translate the pseudocode into Java source code. In the following table, the pseudocode at the left translates into the Java source code at the right.

Programmers normally refer to Java source code instructions as Java *statements*. For Java statements to work, they must use precise syntax. For example, as shown below, Java statements must (1) use a - for subtraction, (2) use a / for division, and (3) end with a semicolon. The precision required by Java statements contrasts with the flexibility of pseudocode. Pseudocode allows any syntax, so long as it is understandable by a person. For example, in pseudocode, it would be acceptable to represent subtraction with a − or the word “subtract.” Likewise, it would be acceptable to represent division with a / or a ÷ or the word “divide.”



Pseudocode	Java Source Code
Calculate ending location minus starting location. Put the result in total distance.	`distanceTotal = locationEnd - locationStart;`
Calculate ending time minus starting time. Put the result in total time.	`timeTotal = timeEnd - timeStart;`
Divide total distance by total time.	`averageMPH = distanceTotal / timeTotal;`

Skipping the Pseudocode Step

Initially, programming language code will be harder for you to understand than pseudocode. But after gaining experience with a programming language, you may become so comfortable with it that you’re able to skip the pseudocode step entirely and go right to the second step, where you write the program using programming language code.

For larger programs, we recommend that you do not skip the pseudocode step. Why? Because with larger programs, it’s important to first focus on the big picture because if you don't get that right, then nothing else matters. And it's easier to focus on the big picture if you use pseudocode where you're not required to worry about syntax details. After implementing a pseudocode solution, it's relatively easy to convert the pseudocode to source code.

1.5 Compiling Source Code into Object Code

After writing a program, you'll want to have a computer perform the tasks specified by the program. Getting that to work is normally a two-step process: (1) Perform a compile command. (2) Perform a run command. When you perform a *compile* command, you tell the computer to translate the program's source code to code that the computer can run. When you perform a *run* command, you tell the computer to run the translated code and perform the tasks specified by the code. In this section, we describe the translation process.

The computer contains a special program called a *compiler* that's in charge of the translation process. If you submit source code to a compiler, the compiler translates it to code that the computer can run. More formally, the compiler compiles the source code and produces *object code* as the result.[2] Object code is a set of binary-format instructions that can be directly run by a computer to solve a problem. An object-code instruction is made up of all 0's and 1's because computers understand only 0's and 1's. Here's an example of an object-code instruction:

```
0100001111101010
```

This particular object-code instruction is referred to as a *16-bit* *instruction* because each of the 0's and 1's is called a *bit,* and there are 16 of them. Each object-code instruction is in charge of only a simple computer task. For example, one object-code instruction might be in charge of copying a single number from some place in main memory to some place in the CPU. There's no need for general-purpose computer programmers to understand the details of how object code works. That's the computer's job, not the programmer's job.

Programmers sometimes refer to object code as *machine code.* Object code is called machine code because it's written in binary, and that's what a computer "machine" understands.

1.6 Portability

In Section 1.2's "Auxiliary Memory" subsection, we said that auxiliary memory is more portable than main memory because it can be moved from one computer to another more easily. In that context, portability referred to hardware. Portability can also refer to software. A piece of software is *portable* if it can be used on many different types of computers.

Portability Problem with Object Code

Object code is not very portable. As you now know, object code is comprised of binary-format instructions. Those binary-format instructions are intimately tied to a particular type of computer. If you have object code that was created on a type X computer, then that object code can run only on a type X computer. Likewise, if you have object code that was created on a type Y computer, then that object code can run only on a type Y computer.[3]

So what's all the fuss about

portability? Who cares that object code is not very portable? Software manufacturers care. If they want to sell a program that runs on different computer types, they typically have to compile their program on the different computer types. That produces different object-code files, and they then sell those files. Wouldn't it be easier if software manufacturers could provide one form of their program that runs on all types of computers?

Java's Solution to the Portability Problem

The inventors of Java attempted to address the inherent lack of portability in object code by introducing the *bytecode* level between the source code and object code levels. Java compilers don't compile all the way down to object code. Instead, they compile down to bytecode, which possesses the best features of both object code and source code:

- Like object code, bytecode uses a format that works closely with computer hardware, so it runs fast.
- Like source code, bytecode is generic, so it can be run on any type of computer.

How can bytecode be run on any type of computer? As a Java program's bytecode runs, the bytecode is translated into object code by the computer's bytecode interpreter program. The bytecode interpreter program is known as the *Java virtual machine,* or *JVM* for short. Figure 1.5 shows how the JVM translates bytecode to object code. It also shows how a Java compiler translates source code to bytecode.

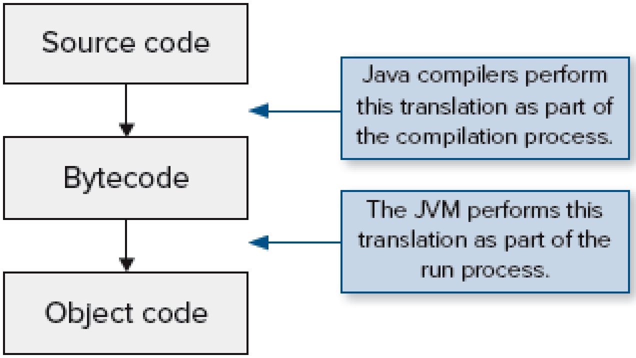

Source code
Bytecode
Object code
Java compilers perform this translation as part of the compilation process.
The JVM performs this translation as part of the run process.

Figure 1.5 How a Java program is converted from source code to object code

To run Java bytecode, a computer must have a JVM installed on it. Fortunately, installing a JVM is straightforward. It's a small program, so it doesn't take up much space in memory. And it's easy to obtain—anyone can download a JVM for free from the Internet. In Section 1.9, we explain how to download a JVM and install it on your own computer.

Why Is the Bytecode Interpreter Program Called a "Java Virtual Machine"?

We'll now explain the origin of the name "Java virtual machine." For programs written with most programming languages, the CPU "machine" runs the program's compiled code. For programs written in Java, the bytecode interpreter program runs the program's compiled code. So with Java, the bytecode interpreter program acts like a CPU machine. But the bytecode interpreter is just a piece of software, not a piece of hardware like a real CPU. Thus, it's a virtual machine. And that's why Java designers decided to call the bytecode interpreter program a Java virtual machine.

1.7 Emergence of Java

Home-Appliance Software

In the early 1990s, putting intelligence into home appliances was thought to be the next “hot” technology. Examples of intelligent home appliances include coffee pots controlled by a computer and televisions controlled by an interactive programmable device. Anticipating a strong market for such items, Sun Microsystems in 1991 funded a team of researchers to work on the secretive “Green Project,” whose mission was to develop software for intelligent home appliances.

An intelligent home appliance’s intelligence comes from its embedded processor chips and the software that runs on those processor chips. Appliance processor chips change often because engineers continually find ways to make them smaller, less expensive, and more powerful. To accommodate the frequent turnover of new chips, the software that runs on them should be extremely flexible.

Originally, Sun planned to use C++

for its home-appliance software, but it soon realized that C++ wasn't sufficiently portable. Rather than write C++ software and fight C++'s inherent portability problems, Sun decided to develop a whole new programming language for its home-appliance software.

Sun's new language was originally named Oak (for the tree that was outside project leader James Gosling's window), but it turned out that Oak was already being used as the name of another programming language. As the story goes, while a group of Sun employees was on break at a local coffee shop, they came up with the name "Java." They liked the name "java" because of the significant role caffeine plays in the lives of software developers. ☺

World Wide Web

When the market for intelligent home-appliance software proved to be less fertile than anticipated, Sun almost pulled the plug on its Java project during the prerelease development phase. Fortunately for Sun (and for all of today's Java lovers), the World Wide Web exploded in popularity. Sun realized that the web's growth could fuel demand for a language like Java, so Sun decided to continue with its Java development efforts. Those efforts bore fruit when they presented Java's first release at the May 1995 SunWorld Conference. Soon thereafter, Netscape, the world's most popular browser manufacturer at the time, announced its intention to use Java in its browser software. With support from Netscape, Java started with a bang and it's been going strong ever since.

The web relies on web pages being downloaded and run on many different types of computers. To work in such a diverse environment, web page software must be extremely portable. You’re probably thinking, Java to the rescue! Actually, that would be a bit of an exaggeration. The web didn’t need rescuing—the web was doing reasonably well even before Java came into the picture, thank you very much. But Java was able to add some much-needed functionality to plain old web pages.

Prior to Java, web pages were limited to one-way communication with their users. web pages sent information to users, but users did not send information to web pages. More specifically, web pages displayed information for users to read, but users did not enter data for web pages to process. When the web community figured out how to embed Java programs inside web pages, that opened the door to more exciting web pages. Java-embedded web pages were able to read and process user input, and that provided users with a more enjoyable, interactive experience.

Java programs in web pages, called *applets,* helped Java grow into one of the leading programming languages in the world. Although applets were fast and versatile, their versatility created security problems, and some of the other types of web-page programs began to exhibit comparable performance and versatility without concomitant security issues. Consequently, Java applets were deprecated in Java release 9 in 2017, and they were removed in Java release 11 in 2018.

A *Java Server Page* (JSP) is a web page that has fragments of a Java program (as opposed to a complete Java program, like an applet) embedded in it. A *servlet* is a Java program that supports a web page but runs on a different computer. Compared to applets, JSPs and servlets make web pages more secure and display them more quickly. JSPs and servlets are written with support from the *Java Enterprise Edition (JavaEE)* platform. Java EE provides an underlying framework of software that enables programmers to write secure, large-scale programs that rely on various technologies such as JSPs, servlets, and databases. A *database* is a collection of data organized in a manner such that its data can be retrieved, updated, or sorted easily.

Java Today

In January 2010, Oracle Corporation acquired Sun Microsystems for a purchase price of $7.4 billion. With the acquisition, Oracle took on oversight of the Java programming language. According to chief executive officer Larry Ellison, owning Java helped Oracle with the fastest growing part of its business—Java middleware for its Oracle Database software.4 *Middleware* is software that connects two other software components that would normally have a hard time communicating with each other. In Oracle's case, its middleware connects its database software with user-interface software. That leads to database software that's easier to use, and ease of use leads to happier customers.

In today's world, the range of computing systems (from largest to smallest) includes the following:

- Cloud computers, which are large central computers accessible to everyone.
- Enterprise computers, which are enterprise-specific large central computers.
- Personal computers (PCs or desktops), which might

synchronize their data with enterprise or cloud computers.

- Notebook and tablet computers, which might synchronize their data with personal computers, enterprise computers, or cloud computers.
- Cell phones.
- Networked controllers, like remotely readable and/or programmable utility meters, thermostats, lighting controls, automobile controls, and so on.

Oracle's business interests include most of the above computing systems in some way. But managing software evolution is expensive, and some evolving aspects of Java, like improving visual displays, are not crucial to Oracle's core interests. Accordingly, in the release of Java 11 in September 2018, Oracle relinquished responsibility for Java's enhanced graphics software—JavaFX. Responsibility for maintaining and upgrading JavaFX is now with a broader-based industry group called *OpenJDK*. As a member of OpenJDK, Oracle retains responsibility for the older parts of Java, but responsibility for JavaFX and other specialized Java features has been distributed to other members of OpenJDK, like *AdoptOpenJDK*.

Thus, Java is diversifying. At large scales, the web and database applications use the previously mentioned Java Enterprise Edition (JavaEE) platform. At small scales, limited-resource devices like mobile phones are constrained in terms of memory capacity and power. Other small-scale devices like embedded controllers are reminiscent of the home-appliance software that got Java started back in the early 1990s. Limited-resource devices typically use the *Java Micro Edition (JavaME)* platform. A Java application that runs on a standard computer—a desktop or a laptop—typically uses the *Java Standard Edition (Java SE)* platform, which is the platform supported by OpenJDK. In this book, we focus on Java SE applications, because Java SE applications are the most general purpose and provide the best environment for learning programming concepts.

Although you will be working with general-purpose applications most easily implemented on a desktop or laptop computer, you will be learning a language that is arguably more versatile than any other computer language. That's because (1) JVM makes Java programs hardware-independent, and (2) Java modules help limit the scope of what's installed to no more than what's needed. Although the venerable C language may still be the language of choice for embedded controllers, if a particular brand of hardware is “not able” to run Java, it may be because the manufacturer does not want to enable access to downloadable free material that competes with material it wants to sell.

1.8 Computer Ethics

Before you learn how to write your first Java program in the next section, you might want to take a moment to ponder the ethical and social implications of programming and using computer software. Such ponderings are part of a large field of study—computer ethics. We don’t pretend that our book will teach you all about computer ethics, but you should at least be aware that the field exists. As you read through this book, if something sparks your interest about a computer ethics issue, we encourage you to do a Google search and see what you can find. Or check out the Computer Ethics Institute's website at http://computerethicsinstitute.org or the Computer Professionals for Social Responsibility's website at http://cpsr.org.

In 1992, the Computer Ethics Institute created the "Ten Commandments of Computer Ethics." In a nod to the original Ten Commandments (from the Bible and the Torah), the Computer Ethics - Institute's commandments use the archaic "Thou shalt" and "Thou shalt not" format. Here are the commandments:[5]

1. Thou shalt not use a computer to harm other people.

2. Thou shalt not interfere with other people's computer work.

3. Thou shalt not snoop around in

other people's computer files.

4. Thou shalt not use a computer to steal.

5. Thou shalt not use a computer to bear false witness.

6. Thou shalt not copy or use proprietary software for which you have not paid.

7. Thou shalt not use other people's computer resources without authorization or proper compensation.

8. Thou shalt not appropriate other people's intellectual output.

9. Thou shalt think about the social consequences of the program you are writing or the system you are designing.

10. Thou shalt always use a computer in ways that ensure consideration and respect for your fellow humans.

Although certainly not as entrenched

as the Golden Rule (“He who has the gold makes the rules”), these commandments are generally well regarded. Note that the commandments suggest behaviors for individual people. Another way to look at computer ethics is to focus less on individual behaviors and more on holistic goals. For example, the goal of *system reliability* is to ensure that software does what it’s supposed to do. The goal of *privacy* is to enable a person or a group of people to hide personal information from the outside world. The goal of *intellectual property protection* is to enable a person or a group of people to protect creations of the mind (e.g., inventions, literary works, or software) so that outsiders are unable to claim them as their own.

1.9 First Program—Hello World

Earlier in this chapter, you learned what it means to compile and run a Java program. But learning by reading only goes so far. It's now time to learn by doing. In this section, you'll enter a Java program into a computer, compile the program, and run it. What fun!

Development Environments

There are different ways to enter a Java program into a computer. You can use an integrated development environment, or you can use a plain text editor. We'll briefly describe the two options.

An *integrated development environment* (IDE) is a rather large piece of software that allows you to enter, compile, and run programs. The entering, compiling, and running are all part of a program's development, and those three functions are integrated together into one environment. Thus, the name - “integrated development environment.” Some IDEs are free and some are quite expensive. We provide tutorials for several popular IDEs on the book’s website.

A *plain text editor* is a piece of software that allows you to enter text and save your text as a file. Plain text editors know nothing about compiling or running a program. If you use a plain text editor to enter a program, you’ll need to use separate software tools to compile and run your program. Note that *word processors,* like Microsoft Word, can be called text editors, but they’re not plain text editors. A word processor allows you to enter text and save your text as a file. But the saved text is not “plain.” When a word processor saves text to a file, it adds hidden characters that provide formatting for the text like line height, color, and so on. And those hidden characters create problems for Java programs. If you attempt to enter a program into a computer using a word processor, your program won't compile successfully, and it certainly won't run.

Different types of computers have different plain text editors. For example, computers that use Windows have a plain text editor called Notepad. Computers that use UNIX or Linux have a plain text editor called vi. Computers that use Mac OS X have a plain text editor called TextEdit. Note: Windows, UNIX, Linux, and Mac OS X are *operating systems*. An operating system is a collection of programs whose purpose is to help run the computer system. In running the computer system, the operating system manages the transfer of information between computer components, like a traffic cop.

For the rest of this section, we'll

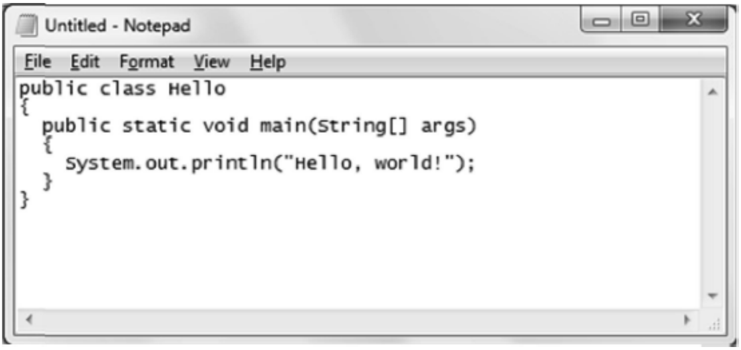
describe how you can enter, compile, and run a program using free, bare-bones tools. You'll use a plain text editor for entering your program, and you'll use simple software tools from Oracle for compiling and running your program. If you have no interest in using such bare-bones tools, and you prefer instead to stick exclusively with an IDE, then refer to the IDE tutorials on the book's website and feel free to skip the rest of this section. If you're unsure what to do, we encourage you to try out the bare-bones tools. They're free and they don't require as much memory as the IDEs. They serve as a standard baseline that you should be able to use on almost all computers.

Entering a Program into a Computer

We'll now describe how you can enter a program into a computer using Notepad, the plain text editor that comes with all versions of Microsoft Windows.

Move your mouse cursor on top of the Start button at the bottom-left corner of your Windows desktop. Click the Start button. (When we ask you to “click” an item, we want you to move your mouse on top of the item and press the left mouse button.) That should cause a menu to appear. On the menu, scroll down to Windows Accessories. Click the **v** symbol to expand that category. Then scroll down to Notepad and click on it. That should cause the Notepad text editor to appear.

In the newly opened Notepad text editor, enter the source code for your first program. More specifically, click somewhere in the middle of the Notepad window and then enter the seven lines of text that are shown in Figure 1.6. When you enter the text, be sure to type the letters with uppercase and lowercase exactly as shown. Specifically, use uppercase for the H's in Hello and the S's in String and System. Use lowercase for all the other letters. Use spaces, not tabs, for indentations. Your entered text comprises the source code for what is known as the Hello World program. The Hello World program is the traditional first program for all programming students. It simply prints a hello message. In Chapter 3, we'll describe the meaning behind the words in the Hello World source code. In this chapter, we're more interested in hands-on experience, and we show you how to enter, compile, and run the Hello World program.



```
public class Hello
{
   public static void main(String[] args)
   {
      System.out.println("Hello, world!");
   }
}
```

Figure 1.6 The Notepad text editor with the Hello World program entered into it

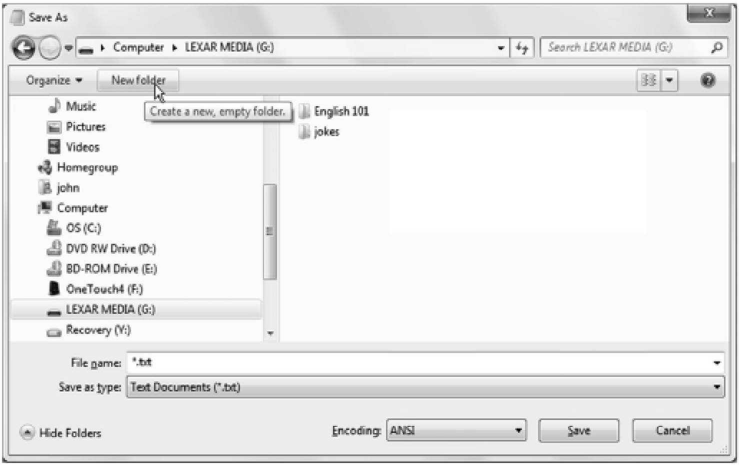
©Oracle/Java

After entering the source code into the Notepad window, you'll need to save your work by storing it in a file. To save your source code in a file, click the File menu in the top-left corner of the Notepad window. That should cause a menu to appear. On the menu, select the Save As option. That should cause a Save As *dialog box* to appear. A dialog box is a small window that performs one task. For this dialog box, the task is to save a file.

Note the File name: box at the bottom of the dialog box. That's where you'll enter the name of your file. But first, you should create a *directory* to store your file in. A directory, also called a *folder,* is an organizational entity that contains a group of files and other directories.6

At the left side of your Save As dialog box, you should see a directory tree. In the directory tree, scroll down until you see a right-facing triangle (▹) with Computer on its right. Click on the triangle to display your computer's contents. Within the now-opened computer container, you should see a right-facing triangle with C: on its right. If you'd like to save on your C: drive, click on C:. If, instead, you'd like to save on a USB flash drive, make sure you have a USB flash drive plugged into your computer, search within your computer container for the drive letter associated with your USB flash drive, and click on that drive letter. Clicking the drive letter should cause the drive letter to appear at the top of your Save As dialog box. Verify that your Save As dialog box now looks similar to the Save As dialog box in Figure 1.7. In particular, note the G: drive at the top of the Save As dialog box. Your drive may be different, depending on what drive letter you clicked.

As shown in Figure 1.7, move your mouse cursor over the New Folder button near the top-left corner of the Save As dialog box. Click the button. That should cause a new directory to appear in the directory tree. The name of the new directory is New Folder by default. The New Folder name should be selected and highlighted. Enter myJavaPgms, and it should overlay the New Folder name. Click the Open button in the bottom-right corner of the dialog box. That should cause your new myJavaPgms directory to appear at the top of your Save As dialog box.


Save As
Computer ▸ LEXAR MEDIA (G:)
Search LEXAR MEDIA (G:)
Organize ▾
New folder
Create a new, empty folder.
Music
Pictures
Videos
Homegroup
john
Computer
OS (C:)
DVD RW Drive (D:)
BD-ROM Drive (E:)
OneTouch4 (F:)
LEXAR MEDIA (G:)
Recovery (Y:)
English 101
jokes
Hide Folders
Save
Cancel

File name: *.txt

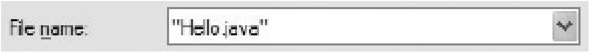
Save as type: Text Documents (*.txt)

Encoding: ANSI

Figure 1.7 Notepad’s Save As dialog box, with the user about to create a new folder

©Oracle/Java

Enter "Hello.java" in the File name: box at the bottom of the dialog box. You must enter "Hello.java" exactly as shown here:


File name: "Hello.java"

©Oracle/Java

Don’t forget the quotes, the uppercase *H*, and the lowercase subsequent letters. Click the Save button in the bottom-right corner of the dialog box. That should cause the Save As dialog box to disappear, and the top of the Notepad window should now say Hello.java. Shut down Notepad by clicking on the X in the top-right corner of the Notepad window.

Installing a Java Compiler and the JVM

In the previous subsection, you entered the Hello World program and saved it to a file. Normally, the next step would be to compile the file. Remember what compiling is? That's when a compiler translates a source code file into a bytecode file. For our Hello World program, the compiler will translate your Hello.java source code file into a Hello.class bytecode file. If you're working in a school's computer lab, chances are pretty good that your computer already has a Java compiler installed on it. If your computer does not have a Java compiler installed on it, you'll need to install it now in order to complete the hands-on portion of this section.

Normally, if someone is interested in installing the Java compiler (to compile Java programs), they are also interested in installing the JVM (to run Java programs). To make the installation easier, Oracle bundles the Java compiler together with the JVM. Oracle calls the bundled software the *Java Development Kit,* or *JDK* for short.

To install the JDK on your computer, go to http://www.mhhe.com/dean3e and click on the Links to Compiler Software link. Then click on the Oracle's Java Download Site link. On Oracle's site, download Java SE and install the JDK as instructed. If you want, you can download and install Oracle's NetBeans IDE at the same time.

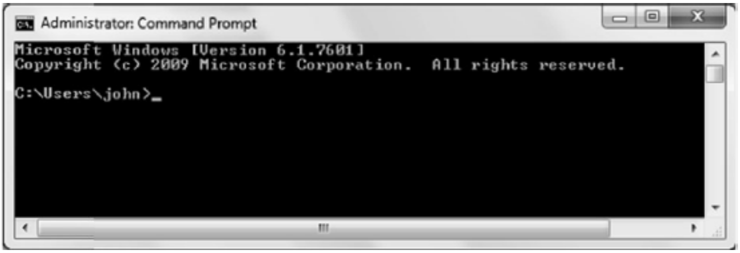
Compiling a Java Program

We'll next describe how you can compile a program using a *command prompt window* (also called a *console*). A command prompt window allows you to enter operating system instructions where the instructions are in the form of words. The words are referred to as *commands*. For example, on a computer that runs the Windows operating system, the command for deleting a file is del (for delete). On a computer that runs the UNIX or Linux operating system, the command for deleting a file is rm (for remove).

To open a command prompt window on a computer that runs the Windows 10 operating system, click the Start button at the bottom-left corner of your Windows desktop. That should cause a menu to appear. On the menu, click the All Programs option. That should cause another menu to appear. In the new menu, click the Accessories directory. That should cause the

Accessories directory to expand and a Command Prompt option to appear. Click the Command Prompt option. That should cause a command prompt window to appear. Figure 1.8 shows the newly opened command prompt window.

In Figure 1.8, note this line:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\john>_
```

Figure 1.8 A command prompt window when it first opens up

©Oracle/Java

```
C:\Users\john
```

That's a *prompt*. In general, a prompt tells you to do something. For a command prompt window, the prompt tells you to enter a command. Very soon, you'll enter commands in your actual command prompt window. But first, note the text at the left of the >

symbol. The text C:\Users\john forms the *path* to the current directory. A path specifies the location of a directory. More specifically, a path starts with a drive letter and contains a series of one or more slash-separated directory names. In our example, C: refers to the hard drive, Users refers to the Users directory that's on the hard drive, and john refers to the john directory that's contained within the Users directory.

To compile your Hello World program, you'll need to go first to the drive and directory where it resides. Suppose your command prompt window's prompt indicates that your current drive is C:, and you saved Hello.java on F:. Then you'll need to change your drive to F:. To do so, enter f: in your command prompt window.

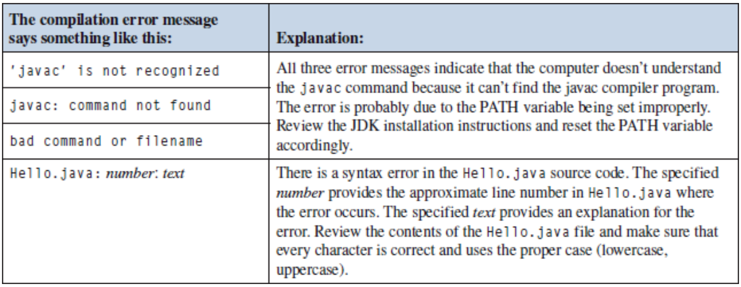
To change to the Hello World program's directory, enter this cd command (*cd* stands for change directory):

```
cd \myJavaPgms
```

Now you're ready to compile your program. Enter this javac command (*javac* stands for java compile):

```
javac Hello.java
```

In entering that command, if your command prompt window displays an error message, refer to Figure 1.9 for possible solutions. If your command prompt window displays no error messages, that indicates success.



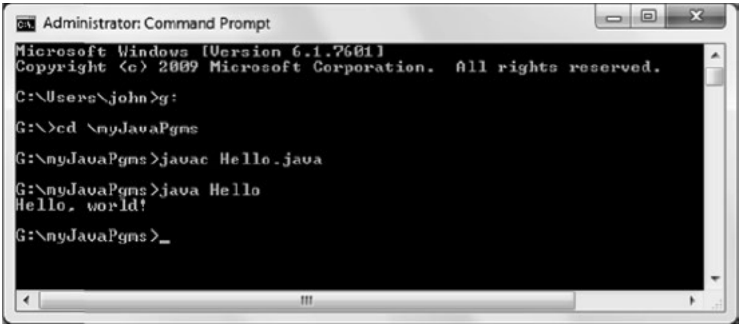
The compilation error message says something like this:	**Explanation:**
`'javac' is not recognized`	All three error messages indicate that the computer doesn't understand the `javac` command because it can't find the javac compiler program. The error is probably due to the PATH variable being set improperly. Review the JDK installation instructions and reset the PATH variable accordingly.
`javac: command not found`	
`bad command or filename`	
`Hello.java:` ***number***: *text*	There is a syntax error in the `Hello.java` source code. The specified ***number*** provides the approximate line number in `Hello.java` where the error occurs. The specified *text* provides an explanation for the error. Review the contents of the `Hello.java` file and make sure that every character is correct and uses the proper case (lowercase, uppercase).

Figure 1.9 Compilation errors and explanations

More specifically, it indicates that the compiler created a bytecode file named Hello.class. To run the Hello.class file, enter this java command:

```
java Hello
```

Your command prompt window should now display your program's output—Hello, world! See Figure 1.10. It shows the command prompt window after completing the steps described previously. Starting with Java 11 you have the option of compiling and executing a non-GUI program with just one command, like this:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\john>g:

G:\>cd \myJavaPgms

G:\myJavaPgms>javac Hello.java

G:\myJavaPgms>java Hello
Hello, world!

G:\myJavaPgms>_
```

Figure 1.10 Compiling and running the Hello World program

```
>java Hello.java
```

However, our compiling and executing examples will not employ this shortcut because we feel that performing the two steps separately reinforces your understanding of the process.

1.10 GUI Track: Hello World (Optional)

This section is the first installment of our optional GUI track. In each GUI-track section, we illustrate a related set of GUI concepts using prewritten GUI software in JavaFX libraries. For example, this section describes two simple ways to display a simple message.

Readers who do not have time for GUI programming can skip the end-of-chapter GUI sections (including this section) and also Chapters 17–19. The GUI in Chapters 17–19 is independent of the GUI in the optional end-of-chapter GUI sections. They are complementary GUI tracks. Depending on your time and interest, you can learn Java from this book with almost any mix of its GUI material. But don't skip all of it. After all, GUI programming is sooooo much fun!

GUI Hello World

This section presents two alternative GUI Hello World programs. Before examining these programs, let's see what they produce. The upper display in Figure 1.11 comes from a program called TitleHello. The lower display comes from a program called LabelHello. In both cases, you see a window with a title bar with standard buttons that move the window temporarily off screen or change its size or close it. Clicking on an empty part of the title bar and moving the mouse drags either window to another part of the screen.

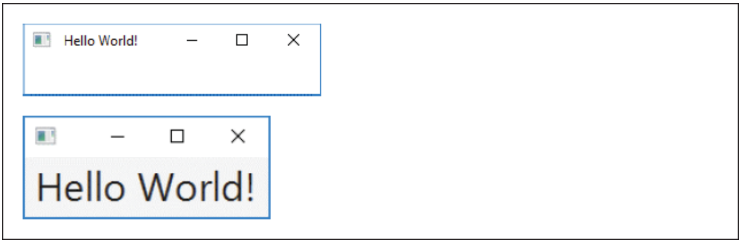


Figure 1.11 Window displays produced by two alternative introductory GUI programs

The upper window comes from the TitleHello program in Figure 1.12.

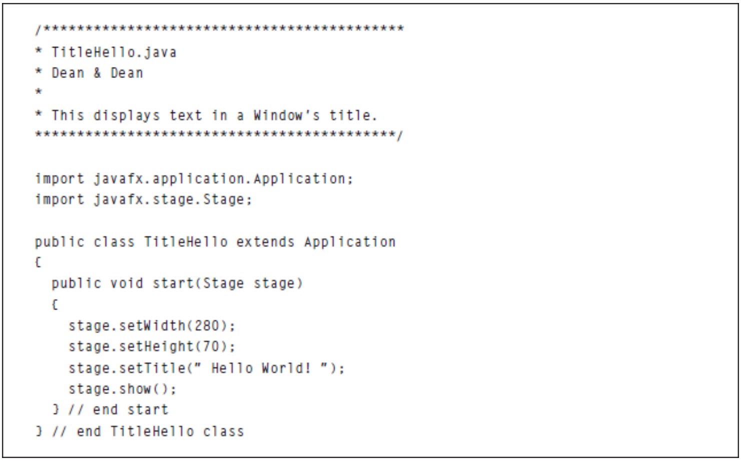
The lower window comes from the LabelHello program in Figure 1.13.

Figure 1.12 contains the code for the TitleHello program in Figure 1.11. This code is slightly more complex than the code in Figure 1.6, which displays output in the command prompt window (Figure 1.10) right after the user's execution command. Some additional complexity is part of the price of GUI.

First, we need to import some of the Java language's prewritten software. In this case we import two prewritten classes:

- Application class in the javafx.application package
- Stage class in the javafx.stage package

Each imported *package* of prewritten software contains a set of related classes and collections of alternative constants called enums (*enum* is short for enumeration). If you want to use more than one class and/or enum from a given package, you can append an asterisk to the package name. For example, the javafx.stage package contains the Stage class and the StageStyle enum. If you want your program to use both of these, you can retrieve them both at the same time by importing the whole package with this one statement: import javafx.stage.*;



```
/*******************************************
* TitleHello.java
* Dean & Dean
*
* This displays text in a Window's title.
*******************************************/

import javafx.application.Application;
import javafx.stage.Stage;

public class TitleHello extends Application
{
  public void start(Stage stage)
  {
    stage.setWidth(280);
    stage.setHeight(70);
    stage.setTitle(" Hello World! ");
    stage.show();
  } // end start
} // end TitleHello class
```

Figure 1.12 TitleHello program that produces the upper output in Figure 1.11

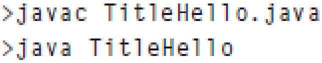
This uses prewritten software in JavaFX's javafx.application and javafx.stage packages. Application and Stage are prewritten classes in these packages.

The extends Application phrase appended to the class header says that the class we are defining will automatically include all of the features already incorporated into the JavaFX's prewritten Application class in addition to the explicit features our code will provide. It turns out that one of the Application class's "features" is a requirement that our program provide a method called start with one parameter of type Stage. Thus, any program that extends Application must define a start method with a parameter of type Stage. Why? Because whenever a JavaFX program like TitleHello executes, underlying software will automatically call its start method and give it an argument of type Stage.

Think of JavaFX's start method like you think of an ordinary Java program's main method. Of course, they are not exactly the same. A main method takes an array of String arguments optionally supplied by the user at the very end of the execution command. A start method takes a Stage argument supplied automatically by prewritten Application software. Every JavaFX program must have a start method, and that's where the programmer's part of the program normally begins execution.

A Stage is a GUI window. JavaFX background activity creates that window and transmits it to our program's start method, which in turn configures it and displays it on the computer screen. Now you can see what the statements inside Figure 1.12's start method do. They set the window's width at 280 pixels, set its height at 70 pixels, give it a text title, and show it on the computer screen. You'll come to understand these and other details better as you progress. Our present objective is to give you a general feeling for how things work.

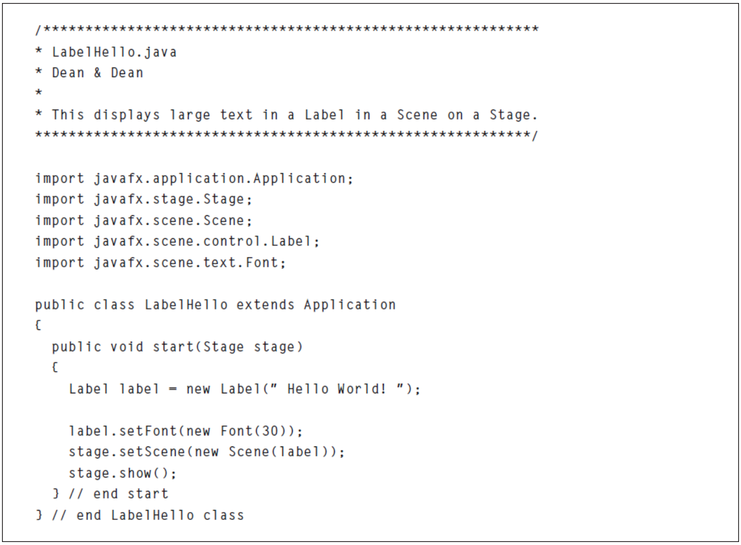
So, go ahead and enter the program code into a text editor. If you need a refresher on how to do that, see the previous section. But this time, instead of saving your source code file with the name Hello.java, save it with the name TitleHello.java. When saving TitletHello.java, make sure you spell the filename with capitals for T and H since that's how TitleHello is spelled in your program's class header. Then, compile and run the program using the console commands:



```
>javac TitleHello.java
>java TitleHello
```

This should work just fine. However, if you try to compile in an IDE, you may get an error message like "Class 'TitleHello' does not have a main method." The next end-of-chapter GUI section will show how one can avoid this possible annoyance.

Figure 1.13 contains the code for the LabelHello program, which produces the lower display in Figure 1.11. This code is more complex than the code in Figure 1.6 and more complex than the code in Figure 1.12. The additional complexity provides more versatility and gives the programmer more power to customize the display.



```
/************************************************************
* LabelHello.java
* Dean & Dean
*
* This displays large text in a Label in a Scene on a Stage.
************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.scene.text.Font;

public class LabelHello extends Application
{
  public void start(Stage stage)
  {
    Label label = new Label(" Hello World! ");

    label.setFont(new Font(30));
    stage.setScene(new Scene(label));
    stage.show();
  } // end start
} // end LabelHello class
```

Figure 1.13 LabelHello program that produces the lower output in Figure 1.11

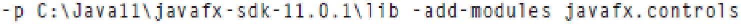
This uses prewritten software in JavaFX's Application, Stage, Scene, Label, and Font classes.

This time, instead of displaying the message in the window's title bar, the program displays the message in the area below the title bar—on the stage where a scene's scenery sits and its actors perform. Evidently, JavaFX designers chose the terms "scene" and "stage" to promote an analogy between GUI programming and playwriting and between GUI displays and performance art.

Inside this program's start method, the first statement creates a label and writes on it the desired "Hello World" text message—with a blank space at either end to improve appearance. To heighten the drama, it specifies a large font size. Then it sets the scene by moving the label onto the stage. And finally, as in the previous TitleHello program, the stage.show(); statement "pulls the curtain" and displays the result on the computer screen.

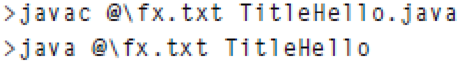
At this point we don't expect you to fully understand constructs like Label, label, and new Scene. However, it's worth noting that capitalized terms like Application, Stage, Scene, Label, and Font identify Java *classes*. Classes are types of things. Uncapitalized terms like stage and label are particular instances of those types. Later, we'll use the term *object* to refer to a particular instance.

In release 11, Java language designers separated JavaFX from the rest of Java. If you download the latest version of Java, you may obtain the two parts from different sources, and you may put them in different places in your computer. Your computer's path variable will probably include the path to (non-GUI) Java, but you may need to identify the path to (GUI) JavaFX explicitly. One way to do this is to create a text file—perhaps called fx.txt – with contents like this:



```
-p C:\Java11\javafx-sdk-11.0.1\lib -add-modules javafx.controls
```

The text string, C:\Java11\javafx-sdk-11.0.1\lib, identifies the path to the JavaFX software on one of our computers. Except for the final \lib, this will probably be different on your computer. Save this fx.txt file at the root of the drive containing your code. Then navigate to the directory containing your code and compile and execute like this:



```
>javac @\fx.txt TitleHello.java
>java @\fx.txt TitleHello
```

Page 28

Summary

- A computer system is all the components that are necessary for a computer to operate and the - connections between those components. More specifically, a computer system consists of the CPU, main memory, auxiliary memory, and I/O devices.
- Programmers write algorithms as first attempt solutions for programming problems.
- Algorithms are written with pseudocode—similar to programming language code except that precise syntax (words, grammar, and punctuation) isn’t

required.

- Source code is the formal term for programming language instructions.
- Object code is a set of binary-encoded instructions that can be executed directly by a computer.
- Most non-Java compilers compile from source code to object code.
- Java compilers compile from source code to bytecode.
- As a Java program runs, the Java virtual machine translates the program’s bytecode to object code.
- Originally, Sun developed Java for use in home-appliance software.
- To expedite development, Java programmers often use integrated development environments, but you can use a plain text editor and

command prompt window.

Review Questions

§1.2 Hardware Terminology

1. What do the following abbreviations mean?

- **a)** I/O
- **b)** CPU
- **c)** RAM
- **d)** GHz
- **e)** MB

2. Identify two important computer input devices.

3. Identify two important computer output devices.

4. Assertions:

- **a)** Main memory is faster than auxiliary memory. (T / F)
- **b)** Auxiliary memory is volatile. (T / F)

c) The first position in main memory is at address 1. (T / F)

d) The CPU is considered to be a peripheral device. (T / F)

e) Hot swapping is when you plug a device into a computer while the computer is on. (T / F)

5. Which type of storage device is normally preferred for tablet computers—hard drive or SSD? Why?

§1.3 Writing Algorithms Using Pseudocode

6. What is an algorithm?

7. What is pseudocode?

§1.4 Translating Pseudocode into Programming Language Code

8. Syntax rules are more lenient for which type of code—

pseudocode or programming language code?

§1.5 Compiling Source Code into Object Code

9. What happens when you compile a program?

10. What is object code?

§1.6 Portability

11. What is a Java virtual machine?

§1.7 Emergence of Java

12. List five different types of Java programs.

Exercises

---

1. [after §1.2] For each of the following computer system components, identify parallel components in or around a human.

- **a)** input devices
- **b)** output devices
- **c)** CPU
- **d)** auxiliary memory

2. [after §1.2] For each of the following items, determine whether it is associated with main memory or auxiliary memory.

- **a)** CD main or auxiliary?
- **b)** solid-state drive main or auxiliary?
- **c)** RAM main or auxiliary?
- **d)** hard drive main or auxiliary?

3. [after §1.2] How many bits in a byte?

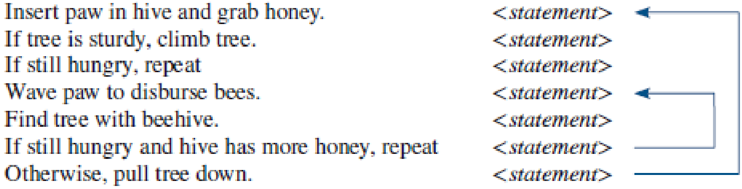
4. [after §1.2] Write the number 9 using 32-bit binary representation.

5. [after §1.2] What type of

computer memory does C: usually refer to?

6. [after §1.2] What is “Moore’s Law”? You won’t find the answer to the question in the book, but you can find it on the Internet. (Hint: Gordon Moore was one of the founders of Intel.)

7. [after §1.3] Rearrange the statements in the left column below to describe an algorithm for a bear gathering honey. Your rearrangement should conform to the diagram in the right column below, with the two arrows emanating from the two repeat statements and pointing to where the respective repetitions begin.



Insert paw in hive and grab honey.
If tree is sturdy, climb tree.
If still hungry, repeat
Wave paw to disburse bees.
Find tree with beehive.
If still hungry and hive has more honey, repeat
Otherwise, pull tree down.

8. [after §1.5] What is the difference between source code and object code?

9. [after §1.6] How is bytecode like source code, and how is it like object code?

10. [after §1.6] What do we call the computer program that converts Java bytecode into a particular type of computer's machine code?

11. [after §1.7] The Java compiler comes in several different configurations or Editions. What is the name of the Edition used for mobile devices and controllers?

12. [after §1.8] According to BBC News, WikiLeaks is a whistle-blowing website "which has established a reputation for publishing sensitive material from governments and other high-profile organizations."

Provide at least one argument in favor of the work done by WikiLeaks. Provide at least one argument in opposition to the work done by WikiLeaks.

13. [after §1.9] Learn how to use TextPad by working your way through the “Getting Started with TextPad” tutorial on the book’s website. Submit hardcopy of the source code for your Countdown program (i.e., print your program from within TextPad). Note that you’re not required to submit source code for your Hello World program or submit output for either program.

14. [after §1.9] (This exercise assumes your computer contains a Hello.java file like that in Figure 1.6. If not, use a plain text editor to create such a file.) On a computer whose operating system is a recent version of Microsoft Windows, invoke:

```
Start > Windows System > Command Prompt
```

Navigate to the directory that has the Hello.java source code. Enter dir Hello.* to list all files starting with “Hello”. If this list includes Hello.class, delete that file by entering del Hello.class. Enter javac Hello.java to compile the source code. Again enter dir Hello.* and verify that the bytecode file, Hello.class, has been created. Now you can enter java Hello to execute the compiled program. Enter type Hello.java and type Hello.class to get a feeling for how bytecode differs from source code.

15. [after §1.9] (This exercise assumes your computer contains a Hello.java file like that in Figure 1.6. If not, use a plain text editor to create such a file.) Experiment with the Hello.java program to learn the meanings of typical compilation and runtime error messages:

- **a)** Omit the final / from the header block.
- **b)** Omit any part of the argument in the parentheses after main.
- **c)** Omit the semicolon from the end of the output statement.
- **d)** One at a time, omit the braces, { and }.
- **e)** Try using lowercase, $, _, or a number for the first character in the class name.
- **f)** Make the program filename different from the class name.
- **g)** Change main to Main.
- **h)** One at a time, try omitting public, static, and void from before main.

Review Question

Solutions

1. What do the following abbreviations mean?

- **a)** I/O: input/output devices
- **b)** CPU: central processing unit or processor
- **c)** RAM: random access memory or main memory
- **d)** GHz: Gigahertz = billions of cycles per second
- **e)** MB: Megabytes = millions of bytes, where 1 byte is 8 bits, and 1 bit is the answer to a single yes/no question

2. The keyboard and a mouse are the two most obvious examples of input devices. Another possible input device is a telephone modem.

3. The display screen and a printer are the two most obvious examples of important output devices. Other examples are a telephone modem and speakers.

4. Assertions:

a) True. Main memory is physically closer to the processor, and the bus that connects the main memory to the processor is faster than the bus that connects the auxiliary memory to the processor. Main memory is also more expensive and therefore usually smaller.

b) False. When power goes off, main memory loses its information, while auxiliary memory does not. An unexpected power failure might corrupt information in auxiliary memory, however.

c) False. The first position in main memory is at address 0.

d) False. The CPU is considered to be part of the computer itself; it's not a peripheral device.

e) True. Hot swapping is when you plug a device into a computer while the computer is on.

5. An SSD is normally preferred for tablet computers, because it is smaller and lighter and more resistant to damage.

6. An algorithm is a step-by-step procedure for solving a problem.

7. Pseudocode is an informal language that uses regular English terms to describe a program's steps.

8. Syntax rules are more lenient for pseudocode (as opposed to programming language code).

9. Most compilers convert source code to object code. Java compilers convert source code to bytecode.

10. Object code is the formal term for binary-format instructions that a processor can read and understand.

11. A Java virtual machine (JVM) is an interpreter that translates Java bytecode into object code.

12. Five different types of Java programs are applets, servlets, JSP pages, micro edition applications, and standard edition applications.

---

[1] "General Motors Replies to Bill Gates," Snopes.com, on the Internet at https://www.snopes.com/fact-check/car-balk.

[2] Most compilers produce object code, but not all. As you'll see in the next section, Java compilers produce an intermediate form of instructions. At a

later time, that intermediate form of instructions is translated into object code.

[3] There are quite a few different computer types that are in common use today. Wikipedia shows 38 computer types, which correspond to 38 categories of CPUs. Each CPU category has its own distinct *instruction set*. An instruction set defines the format and meanings of all the object-code instructions that work on a particular type of CPU. A full discussion of instruction sets is beyond the scope of this book. If you'd like to learn more, see Wikipedia's website and enter "comparison of instruction set architectures" in the search box.

[4] Patrick Thibodeau, "Oracle's Sun Buy: Ellison Praises Solaris, Thumbs Nose at IBM," *ComputerWorld*, April 20, 2009.

[5] Ramon C. Barquin, "In Pursuit of a 'Ten Commandments' for Computer

Ethics," Computer Ethics Institute, May 7, 1992, http://computerethicsinstitute.org/barquinpursuit1992.html.

[6] In the Windows and Macintosh worlds, people tend to use the term *folder*. In the UNIX and Linux worlds, people tend to use the term *directory*. As you'll see in Chapter 16, Oracle uses the term *directory* as part of the Java programming language. We like to follow Oracle, and we therefore use the term *directory* rather than *folder*.

Algorithms and Design

Objectives

- Learn how to write an informal text description of what you want a computer program to do.
- Understand how a flowchart describes what a computer program does.
- Become familiar with the

standard well-structured control patterns.

- Learn how to structure conditional executions.
- Learn how to structure and terminate looping operations, including nested loops.
- Learn how to “trace through” a program’s sequence of operation.
- See how you can describe program operation at different levels of detail.

Outline

2.1 Introduction 32

2.2 Output 33

2.3 Variables 34

2.4 Operators and Assignment Statements 35

2.5 Input 36

2.6 Flow of Control and Flowcharts 37

2.7 if Statements 38

2.8 Loops 43

2.9 Loop Termination Techniques 45

2.10 Nested Looping 48

2.11 Tracing 51

2.12 Problem Solving: Other Pseudocode Formats and an Asset Management Example 55

2.1 Introduction

As indicated in Chapter 1, writing a computer program involves two basic activities: (1) figuring out what you want to do and (2) writing code to do it. You might be tempted to skip the first step and jump immediately to the second step—writing code. Try to resist that urge. Jumping immediately into the code often results in bad programs that work poorly and are hard to fix because poor organization makes them hard to understand. Therefore, for all but the very simplest problems, it's best to start by thinking about what you want to do and then organize your thoughts.

As part of the organization process, you'll want to write an *algorithm*.1 An algorithm is a sequence of instructions for solving a problem. It's a recipe. When specifying an algorithm, two formats are common:

1. The first format is a natural-language outline called *pseudocode*, where the prefix pseudo means fictitious or pretended, so it's not "real" code. Pseudocode, like real code, is composed of one or more

statements. A statement is the equivalent of a "sentence" in a natural language. If the sentence is simple, the corresponding statement usually appears on one line, but if the sentence is complex, the statement may be spread out over several lines. Statements can be nested inside each other, as in an outline. We'll use the term statement a lot, and you'll get a better appreciation for it as we go along.

2. The second format is an arrangement of boxes and arrows that help you visually step through the algorithm. The most detailed form of boxes and arrows is called a *flowchart*. The boxes in a flowchart typically contain short statements that are similar to pseudocode statements.

This chapter shows you how to apply pseudocode and flowcharts to a fundamental set of standard programming problems—problems that appear in almost all large programs. The chapter also shows you how to *trace* an algorithm—step through it one statement at a time—to see what it's actually doing. Our goal is to give you a basic set of informal tools which you can use to describe what you want a program to do. The tools help you organize your thinking before you start writing the actual program. Tracing helps you figure out how an algorithm (or completed program) actually works. It helps you verify correctness and identify problems when things are not right.

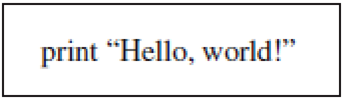
2.2 Output

The first problem to consider is the problem of displaying a program's final result—its output. This may sound like something to consider last, so why consider it first? The output is what the *end user*—the client, the person who eventually uses the program—wants. It’s the goal. Thinking about the output first keeps you from wasting time solving the wrong problem.

Put yourself in user's place.

Hello World Algorithm

In Chapter 1, we showed you a Java program that generated “Hello, world!” output on the computer screen. Now we’ll revisit that problem, but focus on the algorithm, not the program. You may recall that Chapter 1’s Hello World program was seven lines long. Figure 2.1 shows the Hello World algorithm—it contains just one line, a pseudocode print statement. The point of an algorithm is to show the steps necessary to solve a problem without getting bogged down in syntax details. The Hello World algorithm does just that. It shows a simple print statement, which is the only step needed to solve the Hello World problem.



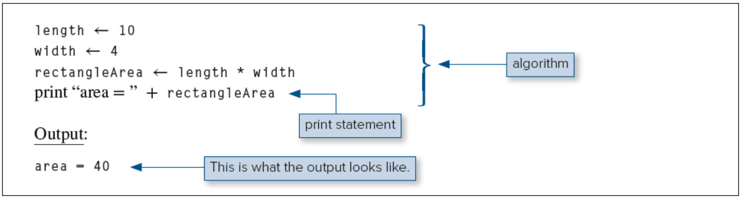
```
print "Hello, world!"
```

Figure 2.1 Hello World algorithm that prints the message "Hello, world!"

Figure 2.1's "Hello, world!" message is a string literal. A *string* is a generic term for a sequence of characters. A *string literal* is a string whose characters are written out explicitly and enclosed in quotation marks. If you print a string literal, you print the characters literally as they appear in the command. So Figure 2.1's algorithm prints the characters H, e, l, l, o, comma, space, w, o, r, l, d, and !.

Rectangle Algorithm

For the next example, suppose you want to display the area of a particular rectangle. First, consider what you want the algorithm to do. In Figure 2.2, look at the output heading and the line underneath it, area = 40. That shows what the algorithm's output looks like.



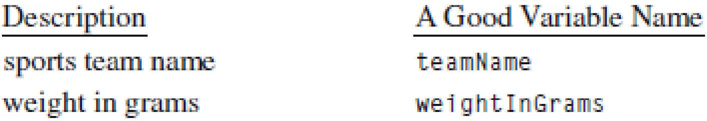
```
length ← 10
width ← 4
rectangleArea ← length * width
print "area = " + rectangleArea

Output:

area = 40
```

Figure 2.2 Rectangle algorithm that prints the area of a rectangle

The top part of Figure 2.2 is the algorithm for calculating a rectangle's area. Note that some of the words, like length and width, appear with monospace font. *Monospace font* is when each character's width is uniform. We use monospace font to indicate that something is a variable. A *variable* is a container that holds a value. To indicate that you want a value to be assigned into a variable, use the backwards arrow symbol, ←, with the variable at the left and the value at the right. For example, the first two lines in Figure 2.2’s algorithm assign 10 to the length variable and 4 to the width variable, respectively. Those statements, and all statements that use the backwards arrow, are pseudocode assignment statements. The Rectangle algorithm’s third line is a more complicated assignment statement. The third line describes two operations: First, compute the area by multiplying length times width. (The * is the multiplication “times” symbol.) Then assign the result (the product) to the variable, rectangleArea. The fourth line prints two items – the string literal “area = ” and the value of the rectangleArea variable. When a variable appears in a print statement, the print statement prints the value stored inside the variable. retangleArea contains 40, so the print statement prints the value 40. In the print statement, note the + symbol separating “area = ” and rectangleArea. We do this to help you develop good habits for later Java coding. Java requires that whenever you use a print statement to print a string plus something else, you insert a + sign between each of the things that you're printing. This chapter is about algorithms and pseudocode, and pseudocode is very forgiving, so if you forget to use the + sign in this chapter to separate a string from another thing that is to be printed, it's not a big deal.

2.3 Variables

Now let's consider variables in more detail. Figure 2.2's Rectangle algorithm has three variables—length, width, and rectangleArea. In rectangleArea, notice how we run together the two words, "rectangle" and "area", and notice how we start the second word with a capital letter. We do this to help you develop good habits for later Java coding (that's the same rationale that we used for inserting a + sign between things that you want to print). Page 35Java does not permit any spaces in a variable name. Therefore, if you'd like to use a variable name with more than one word, don't use spaces to separate the words. Instead, to distinguish the words, we suggest that you capitalize the first letter of all words after the first word. All other letters, including the very first letter, should be lowercase. This technique for writing multiple-word variable names is called *camelCase*, because of the bump(s) in the middle. Here are two examples that show how to name variables with camelCase:



Description	A Good Variable Name
sports team name	`teamName`
weight in grams	`weightInGrams`

Variables can hold different *types* of data. Which type of data would the teamName variable probably hold—a number or a string? It would probably be used to hold a string (e.g., “Jayhawks” or “Pirates”). Which type of data would the weightInGrams variable probably hold—a number or a string? It would probably be used to hold a number (e.g., 12.5). It’s relatively easy for a human to determine the type of a named variable by just thinking about the name, but this kind of thinking is very difficult for a computer. So in a real Java program, we must tell the computer the type of each data item.

However, since pseudocode is designed strictly for humans and not computers, in pseudocode we don’t bother with type specification. Notice that Figure 2.2’s pseudocode representation of a Rectangle program does not contain any mention of data type. Pseudocode ignores data type so that focus can be kept on the algorithm’s essence—its instructions.

2.4 Operators and Assignment Statements

The previous section described variables by themselves. Now let’s consider relationships between variables by looking at operators and assignments.

Here is the third statement from Figure 2.2’s Rectangle algorithm:

```
rectangleArea ← length * width
```

As indicated earlier, the * symbol is the multiplication operator. The other common arithmetic operators are + for addition, − for subtraction, and / for division. These should be familiar to everyone. The length and width variables are *operands*. In mathematics, and again in programming, an operand is an entity (e.g., a variable or a value) that is operated on by an operator. The length and width variables are operands because they are operated on by the * operator.

When we say “variableA ← x,” we mean “put the value of x into variableA” or “assign the value of x to variableA.” So the rectangleArea ← length * width statement puts the product of length times width into the rectangleArea variable. A picture is worth a thousand words. See Figure 2.3—it visually describes what the statement does.

Figure 2.3 includes a pair of parentheses not shown in the pseudocode statement. As you might remember from grade school math, parentheses have the highest *precedence* of all the math operators. That means that any operations inside parentheses (like multiplication or addition) should be performed before operations that appear outside parentheses. So Figure 2.3 indicates that everything on the right side of the backwards arrow Page 36(←) should be evaluated completely before the result is assigned to the variable on the left side of the backwards arrow. That’s the way it is in pseudocode, and that’s also the way it is in Java. You can include the parentheses in your code if you like, but most people don’t bother because they know that the parentheses are implied. The backwards arrow’s direction—right to left—indicates that the assignment statement flows right to left, and that helps to make the implied parentheses more obvious. In other words, with a right-to-left flow, the right side of the backwards arrow must be evaluated before its value can be transferred to the left side of the backwards arrow.

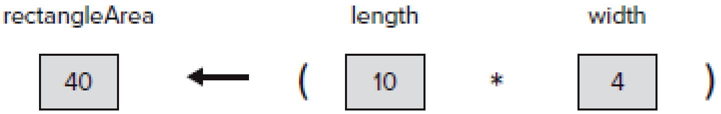
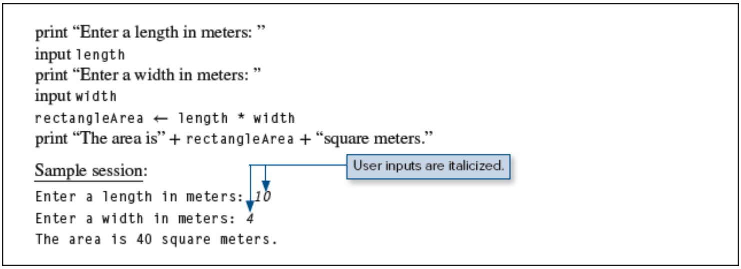


Figure 2.3 Assignment operation represented by left-pointing arrow

2.5 Input

In the preceding Rectangle algorithm, the algorithm itself supplied the values for the length and width variables. We did it that way to make the introductory discussion as simple as possible. Sometimes this is an appropriate strategy, but in this particular case, it's silly, because the algorithm solves the problem only for one particular set of values. To make the algorithm more general, instead of having the algorithm supply the values for length and width, you should have the *user* (the person who runs the program) supply the values. When a user supplies a value(s) for a program, that's called *user input,* or just *input.* Figure 2.4 presents an improved Rectangle algorithm, where input length and input width perform user input operations.

Note the first two print statements in Figure 2.4—they're called *prompts* because they tell (or prompt) the user what to enter. Without prompts, most users would be left with an unpleasant sensation and the puzzling question, "What do I do now?"



```
print "Enter a length in meters: "
input length
print "Enter a width in meters: "
input width
rectangleArea ← length * width
print "The area is" + rectangleArea + "square meters."
```

Sample session:

```
Enter a length in meters: 10
Enter a width in meters: 4
The area is 40 square meters.
```

User inputs are italicized.

Figure 2.4 Rectangle algorithm that gets length and width values from a user

Throughout the book, we provide *sample sessions* as a means of showing what happens when an algorithm or program is run with a typical set of inputs. When there is space, we include the sample session in the figure with the algorithm or program that generates it. Can you identify the user-input values in the sample session in Figure 2.4? Our convention is to italicize sample session input values to distinguish them from output. Thus, 10 and 4 are user-input values.

Write what you'll do and how you'll do it.

The combination of a pseudocode algorithm and a sample session represents a convenient and efficient way to specify a simple algorithm or program. The sample session shows the format of desired inputs and outputs. It also shows representative input and output numerical values, which allow a programmer to verify that his/her completed program actually behaves as required. In many of the book's projects (projects are on the book's website), we provide some combination of pseudocode and sample session to specify the problem we are asking you to solve.

2.6 Flow of Control and

Flowcharts

In the preceding sections, we described various statements—print statements, assignment statements, and input statements—and we focused on the mechanics of how each statement works. Now it's time to focus on the relationships between statements. More specifically, we'll focus on *flow of control*. Flow of control is the order in which program statements are executed. In our discussion of flow of control, we'll refer to both algorithms and programs. Flow of control concepts apply equally to both.

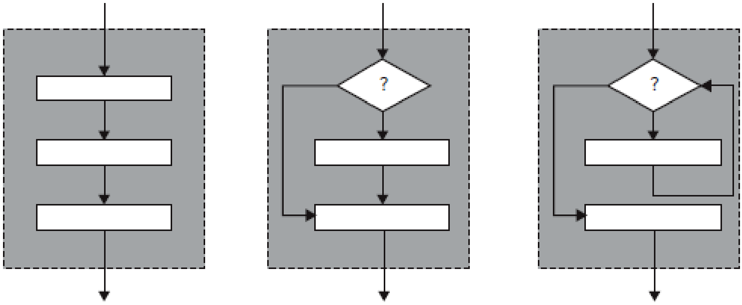
Flow of control is best explained with the help of flowcharts. Flowcharts are helpful because they are pictures. As such, they help you to "see" an algorithm's logic. A flowchart uses two basic symbols: (1) rectangles, which contain commands like print, assign, and input, and (2) diamonds, which contain yes/no questions. At each diamond, the flow of control splits. If the answer is “yes,” flow goes one way. If the answer is “no,” flow goes another way.

The dashed boxes in Figure 2.5 show three standard structures for flow-of-control—a sequential structure, a conditional structure, and a looping structure. The flowchart on the left—the sequential structure—is a picture of the Rectangle algorithm described in Figure 2.2. *Sequential structures* contain statements that are executed in the sequence/order in which they are written. In other words, after executing a statement, the computer executes the statement immediately below it. *Conditional structures* contain a yes/no question, and the answer to the question determines whether to execute the subsequent block of statements or skip it. *Looping structures* also contain a yes/no question, and the answer to the question determines whether to repeat the loop's block of statements or move on to the statements after the loop.

Structured programming is a discipline that requires programs to limit their flow of control to sequential, conditional, and looping structures. A program is considered to be well structured if it can be decomposed into the patterns in Figure 2.5. You should strive for well-structured programs because they tend to be easier to understand and work with. To give you an idea of what not to do, see Figure 2.6. Its flow of control is bad because there are two points of entry into the loop, and when you're inside the loop, it's hard to know what's happened in the past. When a program is hard to understand, it's error-prone and hard to fix. Code that implements an algorithm like this is sometimes called spaghetti code because when you draw a flowchart of the code, the flowchart looks like spaghetti. When you see spaghetti, untangle it!

Don't write spaghetti code.

In addition to standardizing sequential, conditional, and looping control structures, structured programming also splits up large problems into smaller subproblems. In Java, we put the solution to each subproblem in a separate block of code called a *method*. We'll discuss methods in Chapter 5, but for now, we'll focus on the three control structures shown in Figure 2.5.



Sequential Conditional Looping

Figure 2.5 Well-structured flow of control

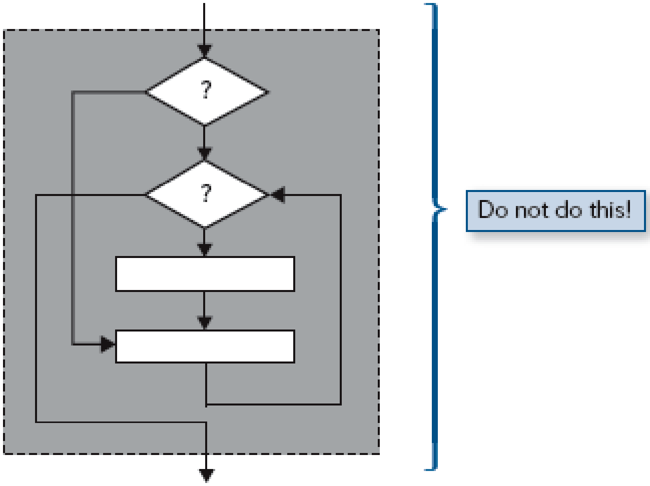


Figure 2.6 Poorly structured flow of control

2.7 if Statements

Page 38

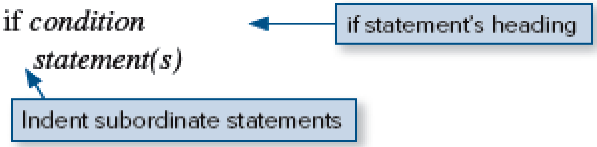
In previous sections describing print, assignment, and input statements, you saw examples of the sequential control structure on the left side of Figure 2.5. Now let's consider the conditional control structure in the center of Figure 2.5. In going through a sequence of steps, sometimes you get to a "fork in the road," at which point you must choose which way to go. The choice you make depends on the situation. More specifically, it depends on the answer to a question. When a program has a fork in the road, programmers use an *if statement* to implement the fork. The if statement asks a question and the answer to the question tells the algorithm which way to go. More formally, the if statement contains a *condition*. A condition is a question whose answer is either yes or no. The answer to the condition's question determines which statement executes next. Here are three forms for the if statement:

"if"
"if, else"
"if, else if"

Now let's look at each of these three forms separately.

"if"

First, suppose you want to do either one thing or nothing at all. In that case, you should use the simple "if" form of the if statement. Here is its format:



```
if condition
   statement(s)
```

Note the italics for "condition" and "statement(s)." Throughout the book, we use italics for items that require a description. Thus, when you see "*condition*," it tells you that an actual condition, not the word "condition," is supposed to follow the word "if." Likewise, when you see "*statement(s)*," it tells you that one or more actual statements, not the word "statements," is supposed to go underneath the if statement's heading.

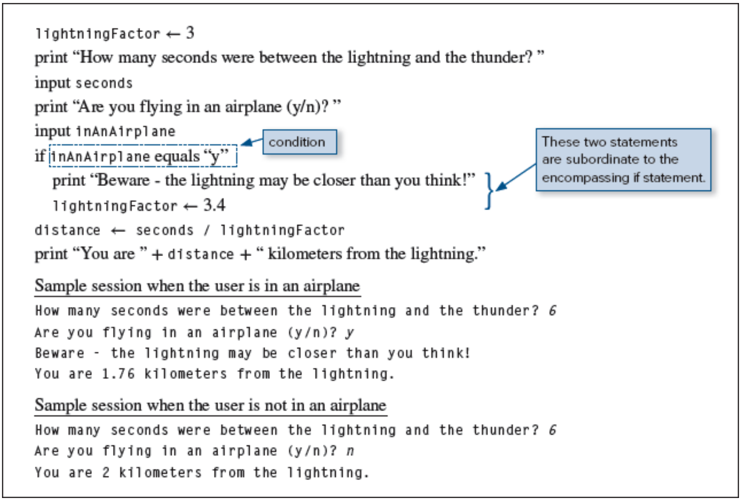
In the above if statement illustration, note how *statement(s)* is indented. Pseudocode emulates a natural-language outline by using indentation to show encapsulation or subordination. The statements under an if statement's heading are subordinate to the if statement because they are considered to be part of the larger, encompassing if statement. Since they are subordinate, they should be indented.

Here's how the simple "if" form of the if statement works:

- If the condition is true, execute all subordinate statements; that is, execute all indented statements immediately below the "if."
- If the condition is false, jump to the line after the last subordinate statement; that is, jump to the first un-indented statement below the "if."

Let's put these concepts into practice by showing an if statement in the context of a complete algorithm.

Figure 2.7's Lightning algorithm determines how far the user is from a lightning strike. The algorithm prompts the user for the number of seconds between seeing a lightning flash and hearing the lightning's associated thunderclap. The standard way to calculate the distance from a lightning strike is to divide the number of seconds by a "lightning factor." The lightning factor divisor is different depending on whether the user is on the ground or in an airplane. The algorithm uses an if statement to distinguish between these two cases. Specifically, in Figure 2.7, note how the algorithm initializes the lightning factor to 3 and then, using an if statement, it changes the lightning factor to 3.4 if the user is in an airplane. After the if statement (i.e., after there are no more indented statements inside the if statement's heading), note how the algorithm divides the entered seconds value by the lightning factor and then prints the resulting distance.[2]



```
lightningFactor ← 3
print "How many seconds were between the lightning and the thunder? "
input seconds
print "Are you flying in an airplane (y/n)? "
input inAnAirplane
if inAnAirplane equals "y"
   print "Beware - the lightning may be closer than you think!"
   lightningFactor ← 3.4
distance ← seconds / lightningFactor
print "You are " + distance + " kilometers from the lightning."
```

condition

These two statements are subordinate to the encompassing if statement.

Sample session when the user is in an airplane

```
How many seconds were between the lightning and the thunder? 6
Are you flying in an airplane (y/n)? y
Beware - the lightning may be closer than you think!
You are 1.76 kilometers from the lightning.
```

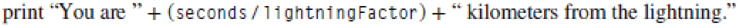
Sample session when the user is not in an airplane

```
How many seconds were between the lightning and the thunder? 6
Are you flying in an airplane (y/n)? n
You are 2 kilometers from the lightning.
```

Figure 2.7 Lightning algorithm that calculates the user's distance from a lightning strike

In the Lightning algorithm, note that the if statement's condition is inAnAirplane equals "y". The inAnAirplane variable holds the user's input, "y" or "n," which indicates whether the user is in an airplane or not. The condition compares the contents of the inAnAirplane variable to the “y” value to see if they’re equal. If they are equal, the if statement’s two indented statements are executed.

There’s one more thing to learn from the Lightning algorithm. Note how the distance assignment command and the subsequent print command are separate statements. That’s perfectly acceptable and quite common, but you should be aware of an alternative implementation where the two commands are merged into one statement:



```
print “You are ” + (seconds / lightningFactor) + “ kilometers from the lightning.”
```

In this case, we put parentheses around the mathematical calculation to emphasize that we want the computer to print the result of the calculation, rather than the values of the individual variables.

“if, else”

Now for the second form of the if

statement—the “if, else” form. Use the “if, else” form if you want to do either one thing or another thing. Here is its format:

```
if condition
   statement(s)
else
   statement(s)
```

And here’s how the “if, else” form of the if statement works:

- If the condition is true, execute all statements subordinate to the “if,” and skip all statements subordinate to the “else.”
- If the condition is false, skip all statement(s) subordinate to the “if,” and execute all statements subordinate to the “else.”

Here’s an example that uses the “if, else” form of the if statement:

```
if grade $\geq$ 60
   print “Pass”
else
   print “Fail”
```

Note how we indent the print “Pass” statement because it is subordinate to the if condition, and we indent the print “Fail” statement because it is subordinate to the “else.”

“if, else if”

The “if, else” form of the if statement addresses situations in which there are exactly two possibilities. But what if there are more than two possibilities? For example, suppose that you want to print one of five possible letter Page 41grades for a particular numerical score. You can do it by using the “if, else if” form of the if statement to establish parallel paths:

```
if grade $\geq$ 90
   print “A”
else if grade $\geq$ 80
   print “B”
else if grade $\geq$ 70
   print “C”
else if grade $\geq$ 60
   print “D”
else
   print “F”
```

What happens if the grade is 85? The print “A” statement is skipped, and the print “B” statement is executed. Once one of the conditions is found to be true, then the rest of the entire if statement is skipped. So the third, fourth, and fifth print statements are not executed.

What happens if all the conditions are false? If all the conditions are false, then the subordinate statement under “else” is executed. So if the grade is 55, print “F” is executed. Note that you’re not required to have an “else” with the “if, else if” statement. If you don’t have an “else” and all the conditions are false, then no statements are executed.

if Statement Summary

Use the first form (“if”) for problems where you want to do one thing or nothing. Use the second form (“if, else”) for problems where you want to do either one thing or another thing. Use the third form (“if, else if”) for problems where there are three or more possibilities.

Use the way
that fits best.

Practice Problem with Flowchart and Pseudocode

Let's practice what you've learned about if statements by presenting a flowchart and having you write the corresponding pseudocode for an algorithm that cuts a CEO's excessively large salary in half. Figure 2.8 shows the flowchart.

In flowcharts, we omit the word "if" from the condition in diamonds and add a question mark to turn the condition into a question. The question format fits well with the "yes" and "no" on the exiting arrows. If the condition is true, the answer to the question is "yes." If the condition is false, the answer to the question is "no." Given the flowchart in Figure 2.8, try to write a pseudocode version of the cut-CEO-salary-in-half algorithm. When you’re done, compare your answer to our answer:

Practice writing a pseudocode algorithm.

```
print “Enter CEO Salary: ”
input ceoSalary
if ceoSalary > 500000
   ceoSalary ← ceoSalary * 0.5
   print “Reduced CEO Salary is $” + ceoSalary
```

In Figure 2.8’s flowchart, note how the diamond’s left arrow and the bottom rectangle’s down arrow are connected. That’s because every flowchart is supposed to have one starting point and one ending point. Don’t forget to connect the bottom components of your flowcharts.

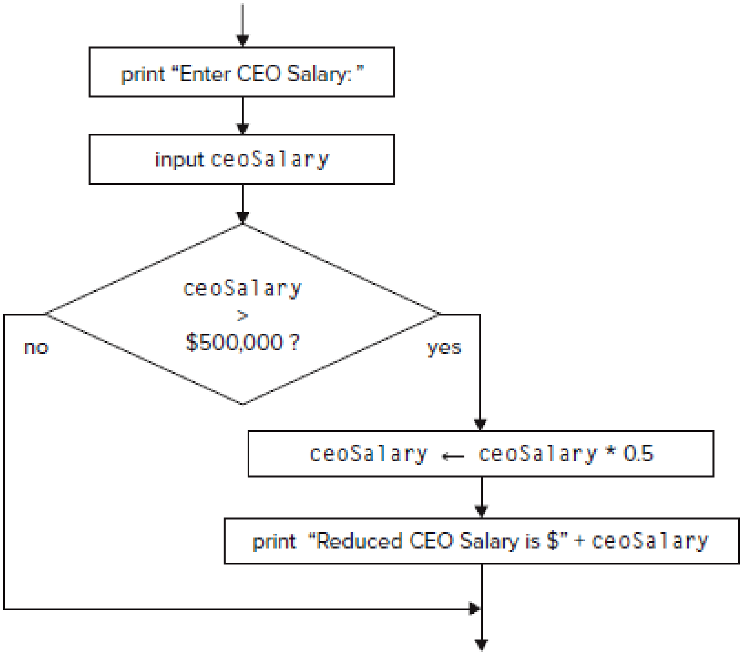
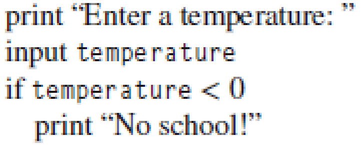

print "Enter CEO Salary: "
input ceoSalary
ceoSalary > $500,000 ?
no
yes
ceoSalary ← ceoSalary * 0.5
print "Reduced CEO Salary is $" + ceoSalary

Figure 2.8 Flowchart for reducing CEO salaries

Practice Problems with Pseudocode Only

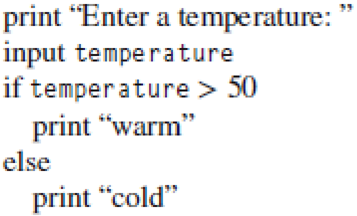
Everybody knows the saying, "A picture is worth a thousand words." This may be true, but compare the space consumed by and the effort to construct Figure 2.8's flowchart with the space consumed by and the effort to write the corresponding pseudocode. Pictures help you get started, but text is more efficient once you know what you're doing. So now let's try skipping the flowchart and going immediately to pseudocode.

First, let's write an algorithm that prints "No school!" if the temperature is below 0 degrees Fahrenheit. Which if statement form should you use for this problem? Because the problem description says to do either something or nothing, you should use the simple "if" form:



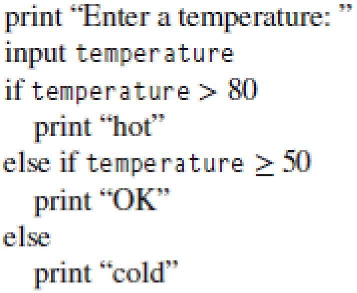
```
print "Enter a temperature: "
input temperature
if temperature < 0
  print "No school!"
```

Next, let's write an algorithm that prints "warm" if the temperature is above 50 degrees and prints "cold" otherwise. Which if statement form should we use? Because the problem description says to do one thing or another thing, you should use the "if, else” form:



```
print "Enter a temperature: "
input temperature
if temperature > 50
   print "warm"
else
   print "cold"
```

Finally, let’s write an algorithm that prints “hot” if the temperature is above 80 degrees, prints “OK” if it’s between 50 and 80 degrees, and prints “cold” if it’s less than 50 degrees. For this problem, it’s appropriate to use the “if, else if” form, like this:



```
print "Enter a temperature: "
input temperature
if temperature > 80
   print "hot"
else if temperature ≥ 50
   print "OK"
else
   print "cold"
```

2.8 Loops

We’ve now discussed two of the three structures in Figure 2.5’sequential structures and conditional structures.

Let’s now discuss the third structure—*looping structures.* Looping structures repeat the execution of a particular sequence of statements. If you need to execute a block of code many times, you could, of course, repeatedly write the code wherever you need it. However, that leads to redundancy, which is something you want to avoid in a computer program because it opens the door to inconsistency. It’s better to write the code once and then reuse it. The simplest way to reuse a block of code is to go back up to before where that block starts, and run through it again. That’s called a *loop*. Every loop has a condition that determines how many times to repeat the loop. Think of driving through western Kansas and seeing a sign for “Prairie Dog Town.” Your kids demand that you take the prairie dog drive-through tour. The decision about how many times to repeat the tour parallels the condition in a loop statement. ⚠

A Simple Example

Suppose you want to print “Happy birthday!” 100 times. Rather than writing 100 print “Happy birthday!” statements, wouldn’t it be better to use a loop? Figure 2.9 presents a solution to this problem in the form of a flowchart with a loop. The flowchart implements the looping logic with an arrow that goes from “count ← count + 1” back up to the “count ≤ 100?” condition.

In a loop, you’ll often use a count variable that keeps track of the number of times the loop has repeated. You can either count up or count down. The Happy Birthday flowchart counts up.

To count up, you’ll need to add 1 to the count variable. In the flowchart, here’s the statement that counts up:

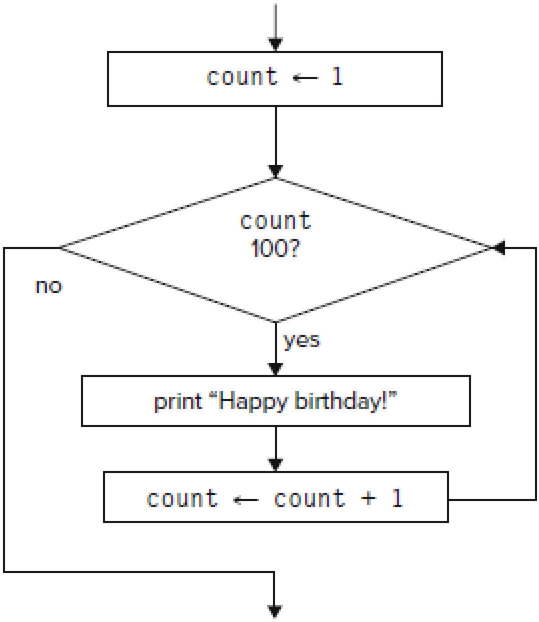
```
count ← count + 1
```

A common mistake by beginning

programmers is to try to count up by using the following statement and nothing else:

```
count + 1
```

That code does indeed add 1 to the current value of count, but then what happens to the sum? It just dies. It isn't saved anywhere. If you ever need to increment a variable, you must do more than just add 1 to the variable. You must use an assignment operator to assign the new value to the variable itself. In pseudocode, we use the backwards arrow for assignment operations, so here, once again, is how you implement the counting part of an algorithm that needs to count (yes, we know we presented the same code 30 seconds ago, but this is so important that we decided to show it twice):


count ← 1
count
100?
no
yes
print "Happy birthday!"
count ← count + 1

`count ← count + 1`

Figure 2.9 Flowchart for the Happy Birthday algorithm

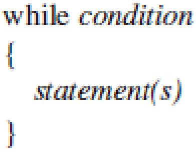
It might look weird to have the same variable on both sides of the backwards arrow, but trust us—it's necessary. It reflects the fact that the statement uses the count variable twice, for different purposes. As you know, assignment statements work right to left, so the count + 1 part of the statement is performed first. Next, the result (the sum of count + 1) is assigned into the count variable, replacing the original count value with the new value.

In practice, all loops should have some kind of termination. That is, they should stop executing at some point. A loop that counts up normally uses a maximum value as a termination condition. For example, Figure 2.9's loop continues so long as count is less than or equal to 100, and it terminates (stops looping) when count reaches 101. A loop that counts down normally uses a minimum value as a termination condition. For example, a loop might start with count equal to 100 and continue so long as count is greater than zero. Then the loop would terminate when count reached zero.

⚠ When a loop's condition compares a counter variable to a maximum value, the question often arises about whether to use "less than or equal to" or just "less than." Likewise, when a loop’s condition compares a counter variable to a minimum value, the question often arises about whether to use “greater than or equal to” or just “greater than.” There are no absolute answers to those questions. Sometimes you’ll need to do it one way, and sometimes you’ll need to do it the other way—it depends on the situation. For example, look again at the decision condition in Figure 2.9’s Happy Birthday algorithm. Suppose you used “less than.” Then, when count equaled 100, you would quit before printing the last (100th) “Happy birthday!” Therefore, in this case you should use “less than or equal to.” If you mistakenly used “less than,” that would be an *off-by-one error*. Such errors are called “off by one” because they occur when you execute a loop one more time than you should or one less time than you should. To avoid off-by-one errors, you should always double-check the borderline cases for your algorithms’ loops.

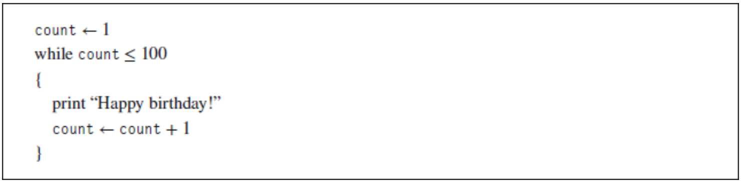
The while Loop

Most popular programming languages have several different types of loops. Although it may be awkward, theoretically, there’s always a way to convert any one type of loop to any other type of loop. So, for simplicity and brevity, in this discussion of algorithms, we’ll consider only one type of loop and look at the other types when we get into Page 45the details of the Java language. The type of loop we’ll consider now is a very popular one, the while loop, which has this format:



```
while condition
{
  statement(s)
}
```

This format should look familiar because it’s similar to the if statement’s format. The condition is at the top, and the subordinate statements are indented. But for the while loop, the subordinate statements are enclosed by braces. The subordinate statements are called the loop's *body*. The number of times that a loop executes its body is called the number of *iterations*. It's possible for a loop to repeat forever, which is called an *infinite loop*. It's also possible for a loop to repeat zero times. There's no special name for the zero-iteration occurrence, but it's important to be aware that this sometimes happens. For an example, let's see how Figure 2.9's Happy Birthday flowchart looks when it's presented as pseudocode with a while loop. This is shown in Figure 2.10.



```
count ← 1
while count ≤ 100
{
  print "Happy birthday!"
  count ← count + 1
}
```

Figure 2.10 Pseudocode for another Happy Birthday algorithm

Here's how the while loop works:

- If the condition is true, execute all of the loop's subordinate statements, and then jump back to the top of the loop to check the loop's condition again.
- When the loop's condition finally becomes false, jump to below the loop (that is, the first statement after the loop's last subordinate statement) and continue execution there.

2.9 Loop Termination Techniques

In this section, we describe three common ways to terminate loops:

- Counter

  Use a counter variable to keep track of the number of iterations.
- User query

  Ask the user if he or she wants to

continue. If the user responds yes, then execute the body of the loop. After each pass through the subordinate statements in the loop, ask the user again if he or she wants to continue.

- Sentinel value

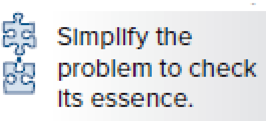
When a loop includes a data-input statement, identify a special value (a sentinel value) that is outside the normal range of input, and use it to indicate that looping should terminate. For example, if the normal range of input is Page 46positive numbers, the sentinel value could be a negative number like -1. Here’s how you do it: Continue to read in values and execute the loop until the entered value equals the sentinel value, and then stop the looping. In the real world, a sentinel is a guard who lets people continue to pass until the enemy arrives. So a program’s sentinel value is like a human sentinel—it allows the loop to continue or not.

Counter Termination

Figure 2.10’s Happy Birthday algorithm is a good example of using a counter to terminate a looping operation. We should point out, however, that the normal place for a computer to start counting is 0, rather than one. If we use the standard start-at-zero convention, Figure 2.10’s pseudocode changes to this:

```
count ← 0
while count < 100
{
  print “Happy birthday!”
  count ← count + 1
}
```

Notice that as we change the initial count value from 1 to 0, we also change condition comparison from ≤ to <. This will produce the same 100 iterations, but this time, the count values will be 0, 1, 2, . . . 98, 99. Each time you create a counter loop, it’s important to assure yourself that the number of iterations will be exactly what you want. Because you can start with numbers different than 1, and because the termination condition can employ different comparison operators, it’s sometimes hard to be sure about the total number of iterations you’ll get. Here’s a handy trick to give you more confidence:



Simplify the problem to check its essence.

To check a loop’s terminal condition, temporarily change the terminal condition to produce what you think will be exactly one iteration. For example, in this most recent pseudocode version of the Happy Birthday algorithm (where the initial count is 0), change the final count from 100 to 1. Then ask yourself, “How many print operations will occur?” In this case, the initial count is

0. The first time the condition is tested, the condition is "0 < 1," which is true. So the condition is satisfied and the loop's subordinate statements execute. Since the final statement in the loop increments the count to 1, the next time the condition is tested, the condition is "1 < 1," which is false. So the condition is not satisfied, and looping terminates. Because using 1 in the loop condition produces one iteration, you can have confidence that using 100 in the loop condition will produce 100 iterations.

User Query Termination

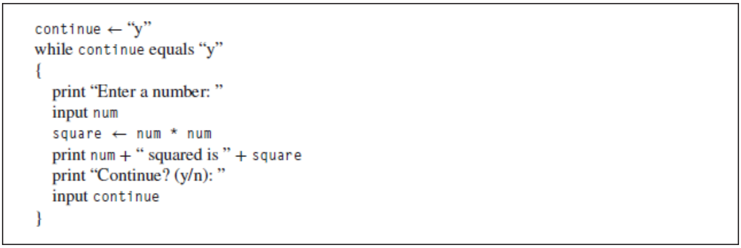
To understand user query termination, consider an algorithm that repeatedly asks a user for numbers and calculates and prints the squares of the input values. This activity should continue so long as the user answers "y" to a "Continue?" prompt.

Figure 2.11 displays this algorithm

as pseudocode. Within the while loop body, the first statement prompts the user to enter a number, the third statement does the computation, and the fourth statement prints the result. The query “Continue? (y/n)” and the corresponding input come just before the end of the body. This loop always executes at least one time, because we assign “y” to the continue variable before the loop begins.

Suppose you want to give the user the opportunity to quit before entering even one number to square. You can do that by replacing the first statement:

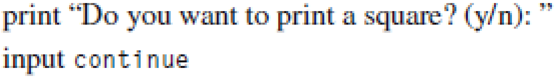
```
continue ← “y”
```



```
continue ← “y”
while continue equals “y”
{
   print “Enter a number: ”
   input num
   square ← num * num
   print num + “ squared is ” + square
   print “Continue? (y/n): ”
   input continue
}
```

Figure 2.11 Print Squares algorithm that uses a query loop

with these two statements:



```
print "Do you want to print a square? (y/n): "
input continue
```

This provides the user the option to enter "n" so that no squares will be computed.

Sentinel-Value Termination

To understand sentinel-value termination, consider an algorithm that reads in bowling scores repeatedly until a sentinel value of −1 is entered. Then the algorithm prints the average score.

Often, you should spend time just thinking about a problem's solution before writing anything down. And you should think first about the solution at a high level, without worrying about all the details. With that said, we encourage you to set the book aside now and think about the steps needed in the Bowling Score algorithm.

Mull it over.

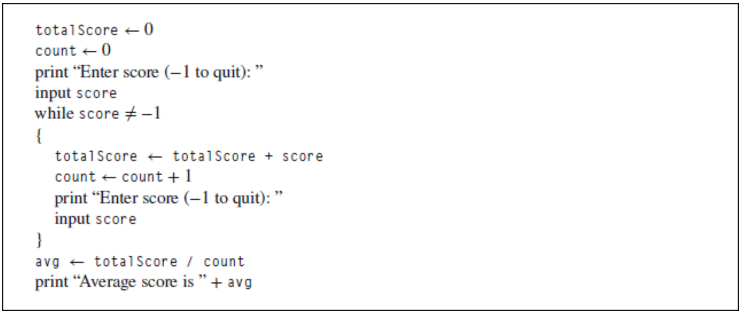
Are you done thinking? If so, compare your thoughts to this high-level description:

> Read in scores repeatedly and find the sum of all the scores.
>
> Then, when −1 is entered, divide the sum by the number of scores entered.

There are two details in this high-level description that you now need to address. First, you need to think about how to find the sum of all the scores. Before asking for any input, and before any looping, assign an initial value of zero to a totalScore variable. In other words, *initialize* it to zero. Then, in the same loop which repeatedly asks the user for the next score, right after inputting that score, add it to the totalScore variable to accumulate the scores as they come in. This way, after all the scores are in, the totalScore variable will already contain the sum of all scores.

The sum of all scores is useful because the goal is to determine the average score, and to compute an average you need the sum. But to compute an average, you also need the total number of items, and that's not known ahead of time. How can you keep track of the number of scores entered so far? Initialize and accumulate a count variable while you initialize and update the totalScore variable. Note that just one loop does all three activities (inputting, updating totalScore, and updating count). We chose −1 as a sentinel value for a Bowling Score algorithm because it's a value that would never be a valid bowling-score entry. But any negative number would work as the sentinel value.

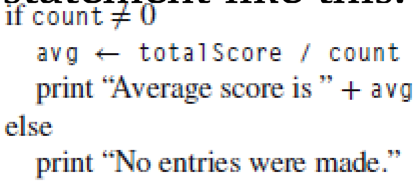
Figure 2.12 illustrates the algorithm solution for this problem. Note how the prompt messages say "(− 1 to quit)." That is necessary because without it, the user wouldn't know how to quit. In general, always provide enough prompting information so that the user knows what to do next and knows how to quit.



```
totalScore ← 0
count ← 0
print "Enter score (−1 to quit): "
input score
while score ≠ −1
{
   totalScore ← totalScore + score
   count ← count + 1
   print "Enter score (−1 to quit): "
   input score
}
avg ← totalScore / count
print "Average score is " + avg
```

Figure 2.12 Bowling Score algorithm using a sentinel-value loop

⚠ What would you expect to happen if the user enters − 1 as the very first input? That causes the loop body to be skipped, and the count variable never gets updated from its original initialized value, 0. When the average assignment statement attempts to calculate the average score, it divides totalScore by count. Because count is 0, it divides by 0. As you may recall from your math courses, division by zero creates problems. If an algorithm divides by zero, the result is undefined. If a Java program divides by zero, the computer prints a cryptic error message and then immediately shuts down the program. Because the Bowling Score algorithm allows for the possibility of division by zero, it is not very *robust*. To be robust, it should behave in a way that a typical user would consider to be both sensible and courteous, even when the input is unreasonable. To make it more robust, replace the last two statements in Figure 2.12's algorithm with an if statement like this:



```
if count ≠ 0
   avg ← totalScore / count
   print "Average score is " + avg
else
   print "No entries were made."
```

Using this if statement enables the program to tell the user why a normal output was not produced, and it avoids the problems inherent with division by zero.

2.10 Nested Looping

In the preceding two sections, we presented algorithms where each algorithm contained one loop. As you proceed through the book and as you proceed through your programming career, you'll find that most programs contain more than one loop. If a program has loops that are independent (i.e., the first loop ends before the second loop begins), Page 49then the program's flow should be reasonably straightforward. On the other hand, if a program has a loop inside a loop, then the program's flow can be harder to understand. In this section, we'll try to make you comfortable with a *nested loop*, which is the formal term for an inner loop that's inside an outer loop.

Suppose you're asked to write an algorithm that finds the largest prime number from a list of user-entered numbers. The user indicates that he or she is done by entering a negative number. For example, suppose the user enters this:

```
5   80   13   21   1   –3
```

The number 5 is prime because it has only two factors, 1 and 5. The number 13 is also prime because it has only two factors, 1 and 13. The numbers 80 and 21 are not prime because they each have more than two factors. The number 1 is not prime by the definition of prime numbers. Thus, the largest prime number in the list is 13.

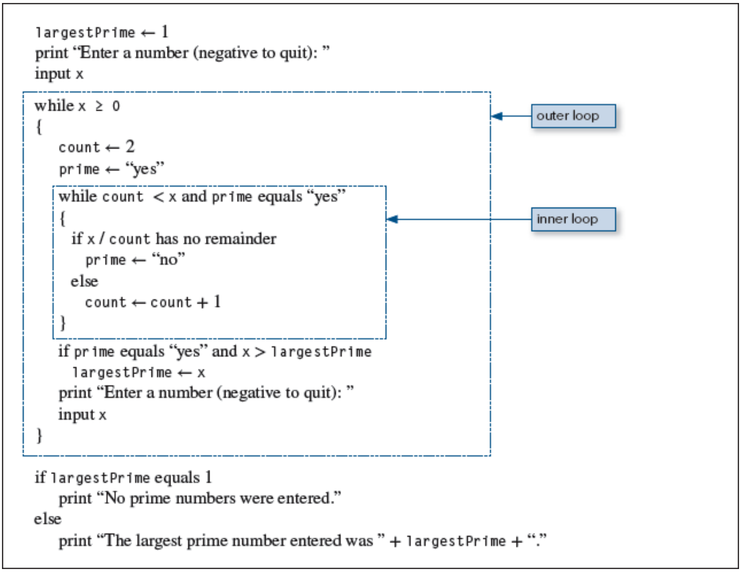
When working on a nontrivial algorithm, before writing anything down, you should think about a very important question: What types of loops should be used? For the find-the-largest-prime-number algorithm, you'll need an outer loop that terminates when the user enters a negative number. What type of loop will that be—a counter loop, user-query loop, or sentinel-value loop? It will be a sentinel-value loop, with a negative number as the sentinel value.

Use most appropriate type of loop.

Figuring out the inner loop is not as easy. Actually, you might not even realize at first that an inner loop is necessary. To determine whether a number x is prime, try to find a number greater than 1 and less than x that divides evenly into x. If you cannot find such a number, then x is prime. So where's the loop? To answer that question, you need to break down your thought process into steps that are simple enough to be part of an algorithm. Specifically, what steps are necessary to find a number greater than 1 and less than x that divides evenly into x? Before you read the answer to that question in the next paragraph, stop and try to answer the question on your own. Hint: You'll need to use a loop.

To find a number greater than 1 and less than x that divides evenly into x, you can use a counter loop that counts from 2 to $x - 1$.[3] For each iteration of the loop, determine whether the count variable's value divides evenly into x. To make that determination, divide x by the count variable and if the remainder is 0, then we say that the count variable's value divides evenly into x. If the count variable's value divides evenly into x, then the count variable's value is a factor of x, and x is not prime. If the count variable gets all the way up to $x - 1$ without finding a factor of x, then x is prime.

Now look at the complete algorithm solution in Figure 2.13. Note that the algorithm does indeed use a sentinel-value outer loop—the loop reads values into a variable named x and terminates when the user enters a negative number. Note that the algorithm does indeed use a counter inner loop – the algorithm initializes a count variable to 2, increments count by 1 inside the loop, and terminates the loop when count reaches the value of x. Now study Figure 2.13's inner loop pseudocode carefully. It does the usual counter loop things, but it includes some additional logic that terminates the loop immediately if it finds a factor of x before count increments all the way up to x. Specifically, the code uses a prime variable to keep track of whether x is a prime number. For each user-entered x value, the algorithm initializes prime to "yes" and then, within the inner loop, changes prime to "no" if it finds that x has a factor. The change to "no" causes the inner loop to terminate.



```
largestPrime ← 1
print "Enter a number (negative to quit): "
input x
while x ≥ 0
{
    count ← 2
    prime ← "yes"
    while count < x and prime equals "yes"
    {
      if x / count has no remainder
        prime ← "no"
      else
        count ← count + 1
    }
    if prime equals "yes" and x > largestPrime
      largestPrime ← x
    print "Enter a number (negative to quit): "
    input x
}
if largestPrime equals 1
    print "No prime numbers were entered."
else
    print "The largest prime number entered was " + largestPrime + "."
```

Figure 2.13 Algorithm that finds the largest prime number from a user-entered list of numbers

Page 50

There's one more important thing to learn from the find-the-largest-prime-number algorithm. The algorithm's goal isn't just to find prime numbers. It's to find the largest prime number from a list of numbers. Finding the largest value from a list of values is a common problem, and you should be familiar with its solution. Let's think about the solution at a high level. After each new prime number is found, the algorithm should ask the question: Is the new prime number larger than the previous largest prime number? If the new prime number is larger, then it becomes the new “champion”; that is, the new largest prime number. Note that the preceding sentence starts with the word *if*. That’s a good indication that you can implement that logic with an if statement. Find the if statement immediately after Figure 2.13’s inner loop and verify that it implements the aforementioned logic. You’ll see that the if statement checks the new number to see if (1) the new number is prime and (2) the new number is larger than the previous largest prime number. If both conditions are met, the algorithm assigns the new number into the largestPrime variable. That assignment crowns the new number as the new champion.

Initialize with a terrible value.

Note the largestPrime

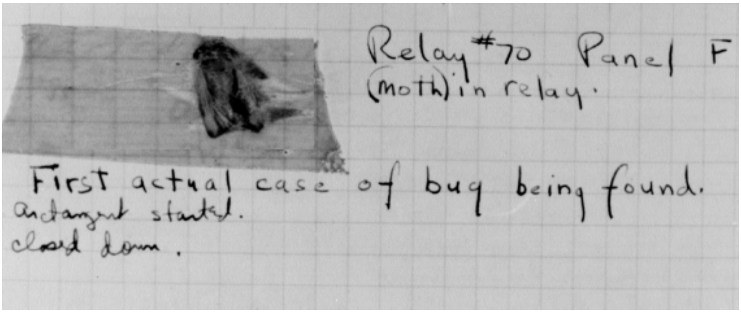
← 1 initialization at the top of the algorithm. What’s the point of initializing largestPrime to 1? You should initialize the champion variable (largestPrime) with a terrible starting value so that it will lose automatically the first time a new prime number is compared to it. You know that 1 will lose to the first entered prime number in a find-the-largest-prime-number contest because Page 51prime numbers are greater than 1 by definition. After the first prime number replaces largestPrime's 1 initial value, subsequent prime numbers might or might not replace largestPrime's value, depending on the value of the new prime number and the value of largestPrime. If the user enters no prime numbers, then largestPrime will retain its original 1 value. The bottom of the algorithm checks for this possibility and prints “No prime numbers were entered” if largestPrime equals 1.

2.11 Tracing

Up to now, we have focused on design. Now let's look at analysis—the breaking up of a whole into its parts. In the present context, that means going through the details of an already-existing algorithm. The analysis technique we'll use is called tracing, where you essentially pretend that you're the computer. You step through an algorithm (or a program) line by line and carefully record everything that happens. In the early parts of this book, we'll use tracing to illustrate programming details that we're trying to explain. Tracing gives you a way to make sure that you really understand newly learned programming mechanisms. Tracing also gives you a way to verify whether an existing algorithm or Java code is correct, or whether it has *bugs*.

Dig into details.

What are bugs, and why are they called that? One of the early digital computers, the Harvard Mark II, used mechanical relays rather than transistors, and programmers programmed by changing electrical connections. As the story goes,[4] even though all the electrical connections were right, the computer kept making a mistake. Finally the programmer discovered a moth squeezed between the contacts of one of the relays. Apparently, the moth had been squashed when the relay contacts closed, and the moth's dead body was interrupting the proper flow of electricity between those contacts. After the programmer pulled the moth out—"debugged" the computer program—the computer gave the right answer. When you're tracing an algorithm or program to find software bugs, you may sometimes feel like one of these old-timers crawling around inside the CPU, looking for moths.



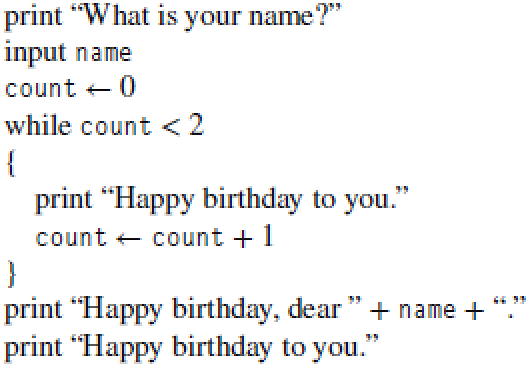
Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.
andangent started.
closed down.

Courtesy of the Naval Surface Warfare Center, Dahlgren, VA., US Naval History and Heritage Command Photograph, 1988.

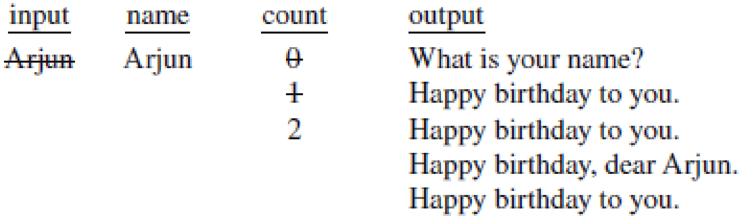
Short-Form Tracing

We present two tracing forms—a short form, described in this subsection, and a long form, described in the next subsection. The short-form tracing procedure is commonly used in industry and in classrooms. It works well in a dynamic environment, where you can move back and forth between pseudocode (or Java code, later) and a trace listing, and fill information in as you go. You may see your teacher go through this dynamic operation on a whiteboard. For example, here’s an algorithm that prints the “Happy Birthday” song:



```
print “What is your name?”
input name
count ← 0
while count < 2
{
   print “Happy birthday to you.”
   count ← count + 1
}
print “Happy birthday, dear ” + name + “.”
print “Happy birthday to you.”
```

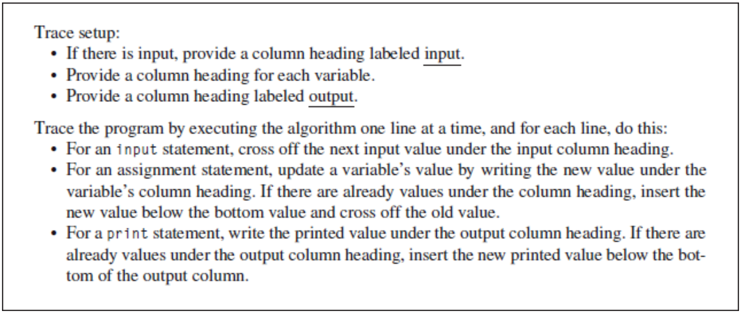
Here’s what the short-form trace looks like after the trace is complete:



input	name	count	output
~~Arjun~~	Arjun	~~0~~	What is your name?
		~~1~~	Happy birthday to you.
		2	Happy birthday to you.
			Happy birthday, dear Arjun.
			Happy birthday to you.

The above trace listing has four columns—input, name, count, and output. The input column shows hypothetical input for the algorithm. The output column shows what the algorithm produces when the algorithm runs with the given input. The name and count columns show the values stored in the name and count variables. In this example, we started with the input value "Arjun." Then we stepped through the code, one line at a time. In stepping through the code, we added values under the name, count, and output columns, and we crossed out old count values as they were overwritten by new count values. Figure 2.14 describes the general procedure.

Page 53



Trace setup:

- If there is input, provide a column heading labeled input.
- Provide a column heading for each variable.
- Provide a column heading labeled output.

Trace the program by executing the algorithm one line at a time, and for each line, do this:

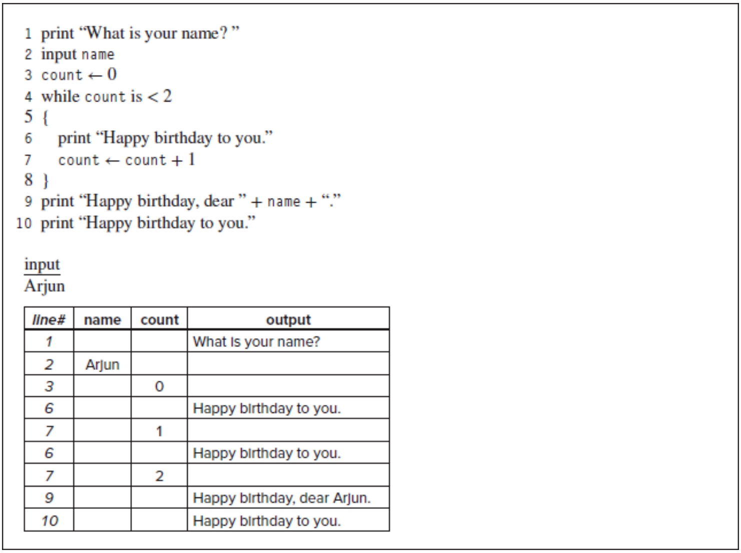
- For an `input` statement, cross off the next input value under the input column heading.
- For an assignment statement, update a variable's value by writing the new value under the variable's column heading. If there are already values under the column heading, insert the new value below the bottom value and cross off the old value.
- For a `print` statement, write the printed value under the output column heading. If there are already values under the output column heading, insert the new printed value below the bottom of the output column.

Figure 2.14 Short-form tracing procedure

Short-form tracing works well in a live interactive context, but it does not work as well in a static context like the pages of a printed book. That’s because in a book, the short-form tracing does not portray the dynamics of the updating process very well. With our simple Happy Birthday algorithm, you may have been able to visualize the dynamics. But for more involved algorithms, a short-form trace listing on the page of a book just “blows through” the details it needs to highlight. Therefore, in this book, we’ll use a long-form tracing procedure that keeps better track of each step as the process unfolds.

Long-Form Tracing

With the long-form tracing procedure, there’s an added emphasis on keeping track of where you are in the algorithm. To implement that emphasis, (1) you need to have a separate row in the tracing table for each step that’s executed in the algorithm, and (2) for each row in the tracing table, you need to provide a line number that tells you the row's associated line in the algorithm. For an example, see the long-form happy birthday trace in Figure 2.15.



```
1  print "What is your name? "
2  input name
3  count ← 0
4  while count is < 2
5  {
6     print "Happy birthday to you."
7     count ← count + 1
8  }
9  print "Happy birthday, dear " + name + "."
10 print "Happy birthday to you."
```

input
Arjun

line#	name	count	output
1			What is your name?
2	Arjun		
3		0	
6			Happy birthday to you.
7		1	
6			Happy birthday to you.
7		2	
9			Happy birthday, dear Arjun.
10			Happy birthday to you.

Figure 2.15 Happy Birthday trace—long form

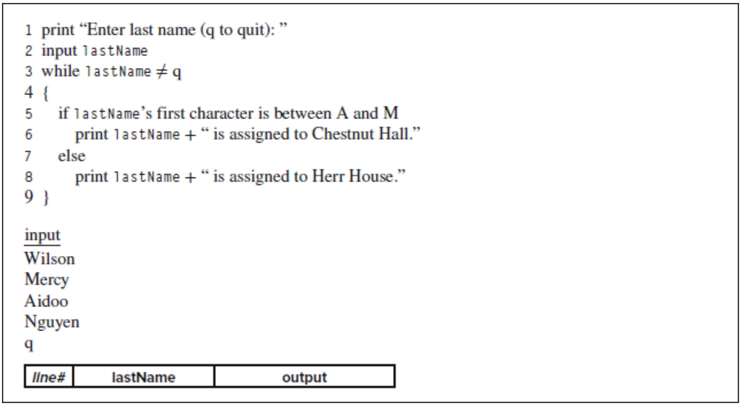
Figure 2.15's long-form trace looks somewhat like the previous short-form trace, with a few notable exceptions. The input column has been moved above the main part of the tracing table. In its place is the line# column, which holds line numbers in the algorithm that correspond to rows in the tracing table. Notice how lines 6 and 7 are executed once, and then executed again. That shows how the trace “unrolls” the loop and repeats the sequence of statements within the loop for each loop iteration.

Using a Trace To Find a Bug

Check each step.

It’s time for you to get your money’s worth from all this tracing talk. We’ll provide you with an algorithm, and it’s up to you to determine whether it works properly. More specifically, trace the algorithm to determine whether each step produces reasonable output. If it produces faulty output, find the algorithm’s bug and fix the algorithm.

Suppose that Park University’s Student Housing office wrote the algorithm shown in Figure 2.16. The algorithm is supposed to read in the names of freshmen and assign each freshman to one of two dormitories. Freshmen with names that begin with A through M are assigned to Chestnut Hall, and freshmen with names that begin with N through Z are assigned to Herr House. Using the trace setup provided in Figure 2.16, try to either complete the trace or get to a point in the trace where you've identified a problem.



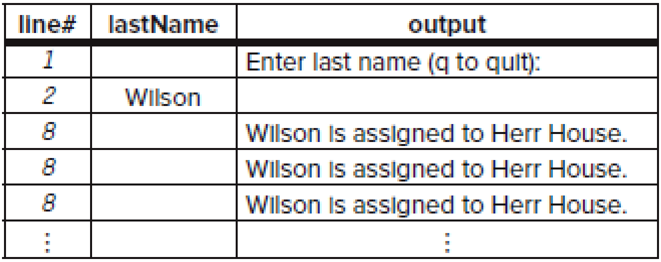
```
1 print "Enter last name (q to quit): "
2 input lastName
3 while lastName ≠ q
4 {
5   if lastName's first character is between A and M
6     print lastName + " is assigned to Chestnut Hall."
7   else
8     print lastName + " is assigned to Herr House."
9 }
```

input
Wilson
Mercy
Aidoo
Nguyen
q

line#	lastName	output

Figure 2.16 Freshmen dormitory assignment algorithm and trace setup

Have you finished working on the trace? If so, compare your answer to this:



line#	lastName	output
1		Enter last name (q to quit):
2	Wilson	
8		Wilson is assigned to Herr House.
8		Wilson is assigned to Herr House.
8		Wilson is assigned to Herr House.
⋮		⋮

The trace points out a problem—the algorithm repeatedly prints Wilson's dorm assignment, but no one else's. There appears to be an infinite loop. Can you identify the bug? The trace shows that lastName gets the first input value, Wilson, but it never gets any other input values. Referring back to Figure 2.16, you can see that the algorithm prompts for the last name above the loop, but not inside the loop. Therefore, the first input value is read in, but no others. The solution is to add another last name prompt inside the while loop, at its bottom. Here is the corrected algorithm: ⚠

```
print "Enter last name (q to quit): "
input lastName
while lastName ≠ q
{
  if lastName's first character is between A and M
    print lastName + "is assigned to Chestnut Hall."
  else
    print lastName + "is assigned to Herr House."
  print "Enter last name (q to quit): "
  input lastName
}
```

We encourage you to trace the corrected algorithm on your own, and you'll find that all four freshmen are assigned to appropriate dorms. Yeah!

Software Development Tools

Most software development tools temporarily label each line of code with a line number to help identify the locations of programming errors. Those line numbers are not actually part of the code, but when they are available, you can use them as identifiers in the *line#* column of a long-form trace. Many software development tools also include a *debugger* that enables you to step through a program one line at a time as it executes. The debugger enables you to look at variable values as you go. Our tracing procedure emulates a debugger's step-by-step type of evaluation. Experience with the tracing used in this book will make it easier for you to understand what an automated debugger is telling you.

2.12 Problem Solving: Other Pseudocode Formats and an Asset Management Example

Pseudocode comes in many different varieties. In this section, we start by describing several alternative pseudocode variations. We then focus on high-level pseudocode, which is particularly helpful for large-scale projects. Finally, we show an example of high-level pseudocode being used for the initial design phase of a large-scale water system asset management project.

Alternative Pseudocode Formats

Up to this point, we've provided one set of constructs for the basic pseudocode commands: the backwards arrow (←) for assignment, the word "equals" for testing for equality, the word "while" for repetition, and so on. The pseudocode constructs in this book are fairly common, but we don't want to give you the impression that they form a standard that is universally followed. On the contrary, pseudocode by its very nature is supposed to be flexible. Its purpose is to provide a mechanism for describing the steps necessary to solve a problem without getting bogged down in syntax details. So if you misspell "while" with two *i*'s (i.e., "whiile"), no problem—

the pseudocode's meaning is still clear. On the other hand, it is important to follow rules that help with clarity—like the rule about indenting when you're inside an if heading or a while loop heading.

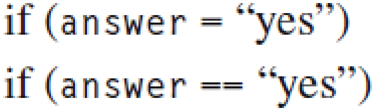
In the interest of consistency, we will stick with the pseudocode constructs that we've presented thus far. However, you should be aware of a few other common pseudocode constructs. For pseudocode assignment, some programmers use the equals sign (=), and some use the words *set* and *to*. For example:

```
x = y
```

```
set x to y
```

Those constructs are acceptable (and you should use them if that's what your teacher likes), but we prefer the backwards arrow for assignment because it emphasizes that the assignment operation flows right to left, with the evaluation on the right side occurring before the result gets transferred to the left.

In pseudocode, to check to see if two entities are equal, some programmers use the equals sign (=) and some use two equals signs (= =). For example:



```
if (answer = “yes”)
if (answer == “yes”)
```

Those constructs are acceptable (and you should use them if that’s what your teacher likes), but we prefer the word *equals* to check to see if two entities are equal because we feel that it’s clearer. Also, if you get in the habit of using = or = = for equality testing in your pseudocode algorithms, your habit might carry over to Java, and that can be problematic. As you’ll learn later, using = to test for equality in Java can lead to difficult-to-detect bugs. And using = = to test for equality in Java can lead to difficult-to-detect bugs if you are comparing strings.

High-Level Pseudocode

⚠ Because pseudocode is so flexible, you can also use it to describe algorithms at a higher, more macroscopic level—with more abstraction. The trick is to ignore the details of subordinate operations and just describe and keep track of inputs to and outputs from those subordinate operations. This strategy presents the “big picture” as seen by the outside world. It looks at the “forest” rather than the “trees.” It helps keep you on the right track—so you don’t solve the wrong problem!

For example, the following Bowling Scores algorithm uses a more high-level pseudocode than what you’ve seen in the past:

```
Input all scores.
Compute average score.
Print the average score.
```

This high-level description presents only the major features, not all the details. It indicates what the program is supposed to do, but not how to do it.

An Asset Management Example

In this subsection, we ask you to think about a real-world managerial problem at a fairly abstract level. Imagine that you are the information technology (IT) specialist working in the government of a small city. The head of that city's water department respects your organizational skills and has asked you to come to a city council meeting and lead a discussion of how you might set up a computer program to help the council manage the assets of that city's water system.

First, you suggest that the city council members help you come up with an overall sequence of steps. On a blackboard, you'll write high-level pseudocode for the “program.” To avoid jargon, you’ll just call this high-level pseudocode a “to-do list.”

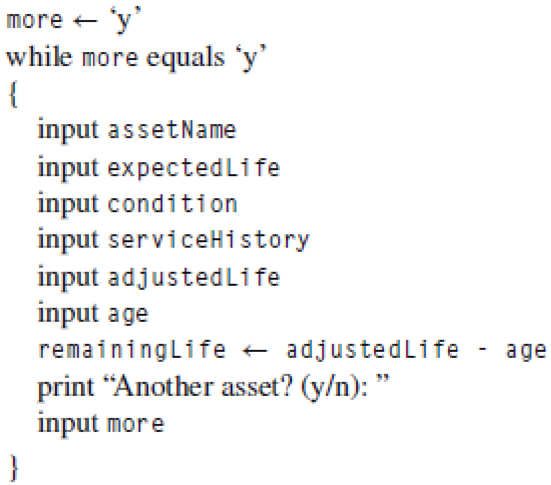
After some discussion, the council members agree on—and you list—the following overall steps:[5]

1. Make an inventory of all water system assets.
2. Prioritize those assets.
3. Schedule future changes, replacements, and additions to those assets.
4. Prepare a long-range budget.

The council thanks you for your help, and for the next meeting, they ask you to flesh out this list with enough detail to show how you plan to implement each of the four steps. They don’t want to see a bunch of Java source code. They just want to see how you’d proceed—to get a feeling for the difficulty of the project.

Back in your office, you get out

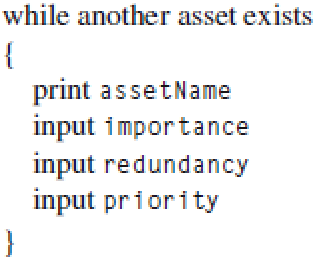
your scratch pad and begin to work. The four steps that you presented in the meeting constitute high-level pseudocode. To “flesh out the list,” you decide to write more detailed pseudocode for each list item. For step 1, you identify seven variables: assetName, expectedLife, condition, serviceHistory, adjustedLife, age, and remainingLife. For each asset, you’ll have to ask someone in the water department to provide appropriate input for each of the first six variables. Then your program will calculate a value for the last variable. You’ll have to repeat this for each significant asset. So here’s an abbreviated pseudocode description of the implementation of step 1:



```
more ← 'y'
while more equals 'y'
{
   input assetName
   input expectedLife
   input condition
   input serviceHistory
   input adjustedLife
   input age
   remainingLife ← adjustedLife - age
   print "Another asset? (y/n): "
   input more
}
```

This algorithm does not include prompts for the individual variables. Some of these variables may have multiple components, and you may wish to establish and enforce certain conventions for what input values will be acceptable. For example, condition and serviceHistory may each have several subordinate components. You'll deal with all those details later.

For step 2, you have five variables: assetName, remainingLife, importance, redundancy, and priority. The assetName and remainingLife variables are the same as two of the variables used for step 1, so you won't need to input those again. But wait! If this is a separate loop, you'll still have to identify each asset to make sure the new values are being associated with the right asset. You could do this by asking the user to re-enter the assetName, or you could do it by looping through all the existing assets and printing out each name just before asking for the required additional information for that asset. The second strategy is easier for the Page 58user, so you pick it. Here's an abbreviated pseudocode description of the implementation of step 2:

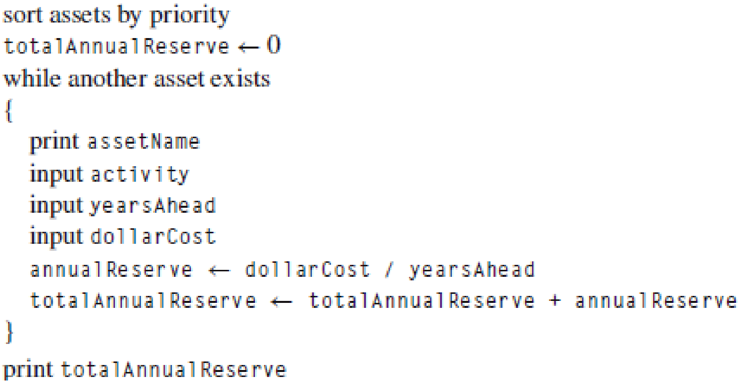


```
while another asset exists
{
   print assetName
   input importance
   input redundancy
   input priority
}
```

Again, the algorithm does not include prompts, and it does not establish and enforce input conventions. You'll deal with those details later.

For step 3, you identify five variables: assetName, activity, yearsAhead, dollarCost, and annualReserve. Again, assetName is already in the system, so again, you can identify it by printing it out. But in scheduling things, the council members will want to deal with the most important things first, so before you start going through the assets, you'll want the program to sort them by priority. The sorting operation might be a little tricky. But if you're lucky, someone else already will have written code for that popular computer task, and you'll be able to use it instead of "reinventing the wheel."

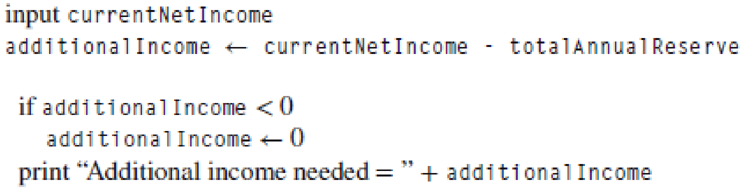
The activity, yearsAhead, and dollarCost are inputs, and you'll want the program to compute annualReserve as dollarCost / yearsAhead. After computing the annual reserve for each individual asset, you'll want the program to add it to a totalAnnualReserve variable, and after the loop, you'll want it to print the final value of totalAnnualReserve. Here's an abbreviated pseudocode description of the implementation of step 3:



```
sort assets by priority
totalAnnualReserve ← 0
while another asset exists
{
    print assetName
    input activity
    input yearsAhead
    input dollarCost
    annualReserve ← dollarCost / yearsAhead
    totalAnnualReserve ← totalAnnualReserve + annualReserve
}
print totalAnnualReserve
```

Again, the algorithm does not include prompts. You'll deal with all those details later.

For step 4, you identify the three variables, totalAnnualReserve, currentNetIncome, and additionalIncome. For this, you need to get someone in the accounting department to provide a value for currentNetIncome. Then have the program subtract it from the totalAnnualReserve computed in step 3 to obtain the additionalIncome required to make the plan work. If the answer comes out negative, you'll want it to just print zero to indicate that your city won't have to come up with any additional income. Here's a pseudocode description of the implementation of step 4:



```
input currentNetIncome
additionalIncome ← currentNetIncome - totalAnnualReserve

 if additionalIncome < 0
   additionalIncome ← 0
 print "Additional income needed = " + additionalIncome
```

OK, that's probably enough preparation for next week's city council meeting. At least you'll be able to give the council members a reasonable sense of the amount of work required.

Summary

- Use pseudocode to write informal descriptions of algorithms. Use understandable names for variables. Indent subordinate statements.

- When your program needs an input, provide an informative prompt to tell the user what kind of information to supply.

- A flowchart provides a visual picture of how the elements of a program are related and how control flows through those elements as the program executes.

- There are three basic well-structured flow-of-control patterns—sequential, conditional, and looping.

- You can implement conditional execution using the three forms of the if statement: “if,” “if, else,” and “if, else if.”

- Provide all loops with some kind of terminating condition such as counter, user query, or sentinel value.

- Use a nested loop if there's a need to repeat something during each iteration of an outer loop.

- Use tracing to (1) obtain an intimate understanding of what an algorithm does and (2) debug programs that have logical errors.

- Use more abstract language to describe larger and more complex programming operations succinctly.

Review Questions

---

§2.2 Output

1. Describe what this statement does:

```
print "user name = " + userName
```

§2.3 Variables

2. Provide an appropriate variable name for a variable that holds the total number of students.

§2.4 Operators and Assignment Statements

3. Write a line of pseudocode that tells the computer to assign distance divided by time into a speed variable.

§2.5 Input

4. Write a line of pseudocode that tells the computer to put a user entry into a variable called height.

§2.6 Flow of Control and Flowcharts

5. What are the three types of control flow described in this chapter?

6. Looping is appropriate

whenever the next thing done is something previously done. (T / F)

§2.7 if Statements

7. Consider the following pseudocode:

```
if it is night, set speedLimit to 55;
otherwise, set speedLimit to 65.
```

Suppose the value of the variable, night, is “false.” After this code runs, what should be the value of the variable, speedLimit?

8. The above pseudocode does not have the exact form suggested in the text. Is that OK?

9. Draw a flowchart that implements this logic:

If the temperature is greater than

10° C and it’s not raining, print “walk.” Otherwise, print “drive.”

10. Provide a solution to the previous problem in the form of pseudocode.

§2.8 Loops

11. Where is a while loop’s terminating decision made?

12. When a while loop terminates, what executes next?

13. Is it possible for a while loop to have an infinite number of iterations?

14. Is it possible for a while loop to have zero iterations?

§2.9 Loop Termination Techniques

15. What are the three loop termination techniques described in this chapter?

16. A *sentinel value* is used to do which of the following?

a) Specify the first value printed.

b) Print an error message.

c) Signal the end of input.

§2.10 Nested Looping

17. How does the form of pseudocode we use in most of this chapter differentiate an inner loop from an outer loop?

§2.11 Tracing

18. Which of the following is true?

a) Tracing shows sequence of execution.

b) Tracing helps you debug a program.

c) Tracing highlights errors in loop initialization and termination.

d) All of the above.

19. Trace the following Bowling Score algorithm (taken from Section 2.9). Use the setup shown below the algorithm.

```
1  totalScore ← 0
2  count ← 0
3  print "Enter score (–1 to quit): "
4  input score
5  while score ≠ –1
6  {
7     totalScore ← totalScore + score
8     count ← count + 1
9     print "Enter score (–1 to quit): "
10    input score
11 }
12 avg ← totalScore / count
13 print "Average score is " + avg
```

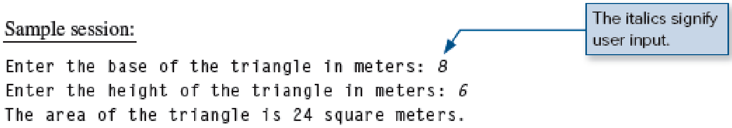
Trace setup:

input
94
104
114
–1

line#	score	totalScore	count	avg	output

Exercises

1. [after §2.5] Write pseudocode for an algorithm that (1) asks the user to input the base and height of a triangle in meters, (2) computes the triangle's area, and (3) prints the triangle's area. Use the following sample session.



Sample session:

```
Enter the base of the triangle in meters: 8
Enter the height of the triangle in meters: 6
The area of the triangle is 24 square meters.
```

The italics signify user input.

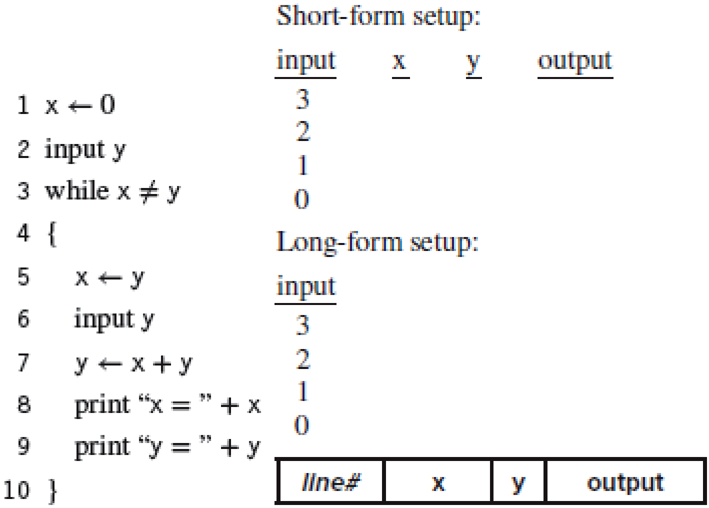
2. [after §2.6] Some programming languages permit arbitrary flow of control from any place in a program to any other place in the program. Not Java. What is the rationale for Java not allowing that?

3. [after §2.8] Given the following pseudocode, circle the statements that are considered to be within the body of the while loop:

```
input timeRemaining
while timeRemaining > 3
{
  print timeRemaining
  timeRemaining ← timeRemaining − 1
}
```

4. [after §2.9] In exercise 3, suppose the user's input for timeRemaining is 10. How many lines of output will the algorithm generate?

5. [after §2.11] Trace the following algorithm. The book presents two ways to do tracing—a short form and a long form. To give you a head start, the setup for the short form and also the long form are given below. For your answer, pick one setup and use it. Skip the other setup.



```
1  x ← 0
2  input y
3  while x ≠ y
4  {
5     x ← y
6     input y
7     y ← x + y
8     print "x = " + x
9     print "y = " + y
10 }
```

Short-form setup:

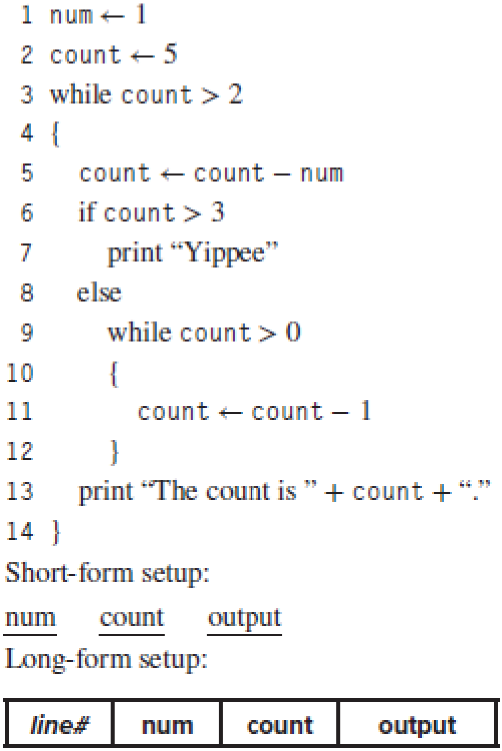
input	x	y	output
3			
2			
1			
0			

Long-form setup:

input
3
2
1
0

line#	x	y	output

6. [after §2.11] Trace the following algorithm. The book presents two ways to do tracing—a short form and a long form. To give you a head start, the setup for the short form and also the long form are given below. For your answer, pick one setup and use it. Skip the other setup.



```
 1 num ← 1
 2 count ← 5
 3 while count > 2
 4 {
 5    count ← count – num
 6    if count > 3
 7       print "Yippee"
 8    else
 9       while count > 0
10       {
11          count ← count – 1
12       }
13    print "The count is " + count + "."
14 }
```

Short-form setup:

num   count   output

Long-form setup:

line#	**num**	**count**	**output**

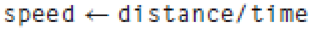
Review Question Solutions

1. The statement prints what is in quotation marks literally, and then prints the current value of the variable userName.

2. totalNumberOfStudents

3. Pseudocode that tells the computer to assign distance divided by time into a speed variable:

Page 63



```
speed ← distance/time
```

4. Pseudocode statement:

input height

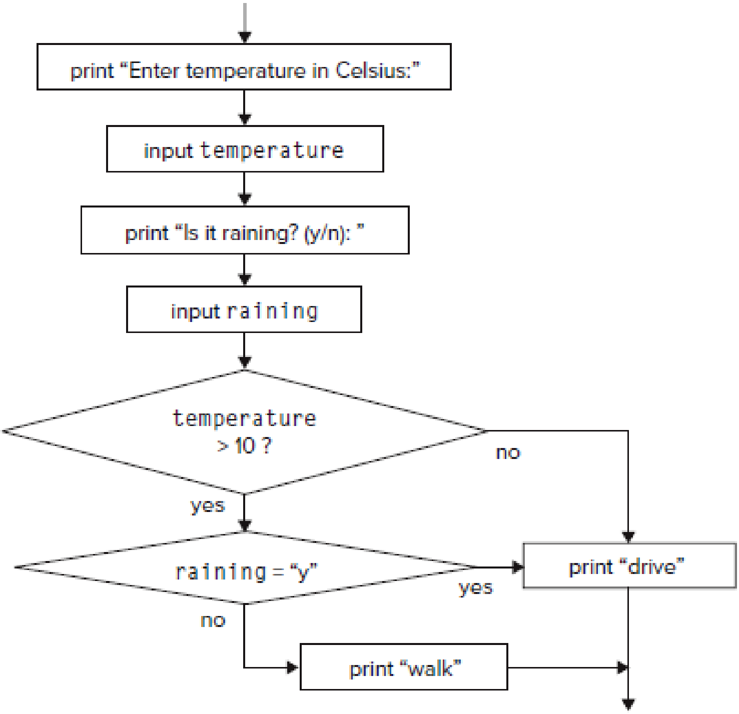
5. The three types of control flow discussed in Chapter 2 are sequential, conditional, and looping.

6. True. Looping is appropriate whenever the next thing done is something previously done.

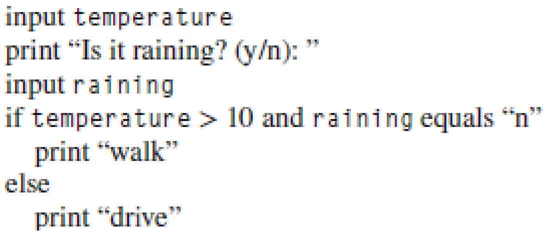
7. After the code executes, the value of the variable, speedLimit, should be 65.

8. Yes. It’s OK because it’s only pseudocode, and it conveys the meaning unambiguously. However, if it were supposed to be code the computer could compile, the syntax would have to conform exactly to prescribed rules for a particular programming language like Java.

9. Flowchart that implements walk/drive logic:


print "Enter temperature in Celsius:"
input temperature
print "Is it raining? (y/n): "
input raining
temperature > 10 ?
no
yes
raining = "y"
yes
print "drive"
no
print "walk"

10. Provide a solution to the previous problem in the form of pseudocode.



```
print "Enter temperature in Celsius: "
input temperature
print "Is it raining? (y/n): "
input raining
if temperature > 10 and raining equals "n"
   print "walk"
else
   print "drive"
```

11. A while loop's terminating decision is made at the beginning of the loop.

12. After a while loop terminates, the next thing to execute is the first statement after the end of the loop.

13. Yes.

14. Yes.

Page 64

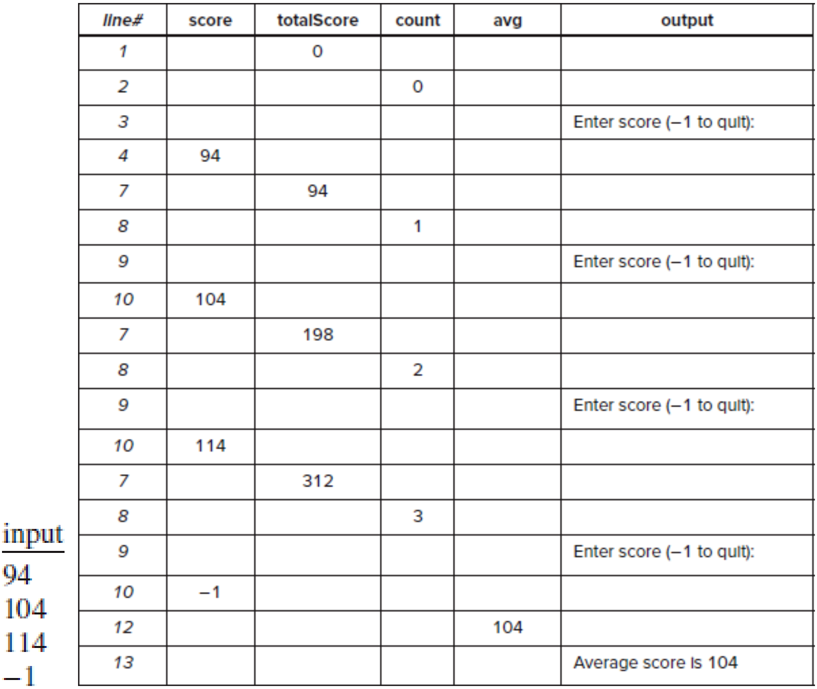
15. The three loop termination techniques described in this chapter are: counter, user query, and sentinel value.

16. A sentinel value is used to: c) signal the end of input.

17. The inner loop is entirely inside the outer loop. The entire inner loop is shifted to the right compared to the outer loop.

18. d) All of above. Tracing shows sequence of execution, helps debug, and highlights initialization and termination errors.

19. Bowling score algorithm trace:



line#	score	totalScore	count	avg	output
1		0			
2			0		
3					Enter score (–1 to quit):
4	94				
7		94			
8			1		
9					Enter score (–1 to quit):
10	104				
7		198			
8			2		
9					Enter score (–1 to quit):
10	114				
7		312			
8			3		
9					Enter score (–1 to quit):
10	–1				
12				104	
13					Average score is 104

input
94
104
114
–1

[1] Ninth-century Persian mathematician Muhammad ibn Musa al-Khwarizmi is considered to be the father of algebra. The term *algorithm* comes from Algoritmi, which is the Latin form of his shortened name, al-Khwarizmi.

[2] For the case of being in an airplane, the lightning-factor divisor is greater, so the calculated distance is less. Why is the calculated distance less when you're in an airplane? When flying in an airplane at a typical cruising altitude of 10,000 meters (= 33,000 feet), the air temperature is around −57°C (= −70°F). Sound travels more slowly in cold air, so in an airplane, when you hear thunder, it's closer than the delay would normally indicate.

[3] Actually, you need to check for factors only from 2 up to the square root of x. If you find a factor greater than the square root of x, it will have a "partner factor" that is less than the

square root of x, such that the product of the two factors equals x. Thus, if numbers less than the square root of x are checked before numbers greater than the square root of x, then finding a factor greater than the square root of x will reveal nothing new.

[4] https://www.computerhistory.org/tdih/september/9

[5] These four steps and their subsequent elaboration are based on recommendations in *Asset Management: A Handbook for Small Water Systems,* Office of Water (4606M) EPA 816-R-03-016, https://nepis.epa.gov/Exe/ZyPDF.cgi?Dockey=P100U7T2.txt, September, 2003.

Java Basics

Objectives

- Write simple Java programs.
- Learn about style issues such as comments and readability.
- Declare, assign, and initialize variables.
- Understand primitive data types

—integer, floating point, and character.

- Understand reference variables.
- Use the String class's methods for string manipulation.
- Use the Scanner class for user input.
- Optionally, learn how to use GUI dialog boxes for user input and output.

Outline

3.1 Introduction 66

3.2 "I Have a Dream" Program 66

3.3 Comments and Readability 67

3.4 The Class Heading 69

3.5 The main Method's

Heading 69

3.6 Braces 70

3.7 System.out.println 71

3.8 Compilation and Execution 73

3.9 Identifiers 73

3.10 Variables 74

3.11 Assignment Statements 75

3.12 Initialization Statements 77

3.13 Numeric Data Types—int, long, float, double 78

3.14 Constants 80

3.15 Arithmetic Operators 83

3.16 Expression Evaluation and Operator Precedence 86

3.17 More Operators: Increment, Decrement, and Compound Assignment 88

3.18 Tracing 90

3.19 Type Casting 90

3.20 char Type and Escape Sequences 93

3.21 Primitive Variables Versus Reference Variables 95

3.22 Strings 96

3.23 Input—the Scanner Class 100

3.24 Simple File Input for Repetitive Testing During Program Development 105

3.25 GUI Track: Input and Output with Dialog Boxes (Optional) 107

3.1 Introduction

In solving a problem, it's best to spend time first thinking about what you want to do and organizing your thoughts. In Chapter 2, you focused on the thinking and organizing by writing pseudocode algorithm solutions for given problem descriptions. In this chapter, you'll take the next step—you'll focus on writing solutions using a real programming language, Java. By using a real programming language, you'll be able to run your program on a computer and produce results on a computer screen.

As you progress through this chapter, you'll find that much of Java's code parallels pseudocode. The primary difference is the precise syntax required for Java. Pseudocode syntax is lenient: Pseudocode must be clear enough so that humans can understand it, but the spelling and grammar need not be perfect. Programming-code syntax is stringent: It must be perfect in terms of spelling and grammar. Why? Because regular programming code is read by computers, and computers are not able to understand instructions unless they're perfect.

Because this chapter is your first real taste of Java, we'll stick to the basics. We'll present Java syntax that's needed for simple *sequential-execution* programs. A sequential-execution program is one in which all the program's statements are executed in the order in which they are written. As we write such programs, we'll show you output, assignment, and input statements. In addition, we'll describe data types and arithmetic operations. Toward the end of the chapter, we'll present a few slightly more advanced topics—type casting and string methods—that will add important functionality without adding much complexity. Let us begin the Java journey.

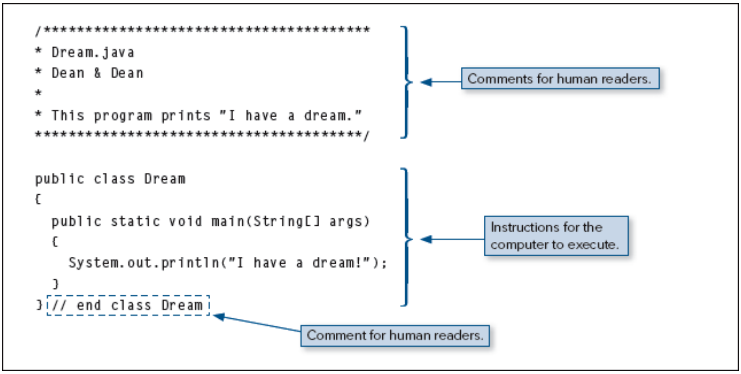
3.2 "I Have a Dream"

Program

In this section, we present a simple program that prints a single line of text. In the next several sections, we'll analyze the different components of the program. The analysis may be a bit dry, but bear with us. It's important to understand the program's components because all future programs will use those same components. In the rest of the chapter, we'll introduce new concepts that enable us to present more substantial programs.

Start every program with this code's structure.

See Figure 3.1. It shows a program that prints "I have a dream!"[1] In the upcoming sections, we'll refer to it as the Dream program. The program contains comments for human readers and instructions for the computer to execute. We'll analyze the comments first, and then we'll move on to the -instructions. You can use this tiny program as a common starting point for all other Java programs. Enter it, run it, and see what it does. Modify it, run it again, and so on, until you have what you need.



```
/****************************************
* Dream.java
* Dean & Dean
*
* This program prints "I have a dream."
****************************************/

public class Dream
{
  public static void main(String[] args)
  {
    System.out.println("I have a dream!");
  }
} // end class Dream
```

Comments for human readers.

Instructions for the computer to execute.

Comment for human readers.

Figure 3.1 Dream program

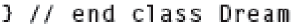
Page 67

3.3 Comments and Readability

In the real world, you'll spend a lot of your time looking at and fixing other people's code. And other people will spend a lot of their time looking at and fixing your code after you've moved on to something else. With all this looking at other people's code going on, everyone's code needs to be understandable. One key to - understanding is good comments. *Comments* are words that humans read but the compiler[2] ignores.

One-Line-Comment Syntax

There are two types of comments—one-line comments and block comments. If your comment text is short enough to fit on one line, use a one-line comment. One-line comments start with two slashes. Here's an example:



```
} // end class Dream
```

The compiler ignores everything from the first slash to the end of the line. So in the above line, the compiler pays attention only to the right brace

(}) and ignores the rest of the line. Why is the comment helpful? If you're viewing a long piece of code on a computer screen and you've scrolled to the bottom of the code, it's nice to see a description of the code (e.g., end class Dream) without having to scroll all the way back up to the beginning of the code.

Block-Comment Syntax

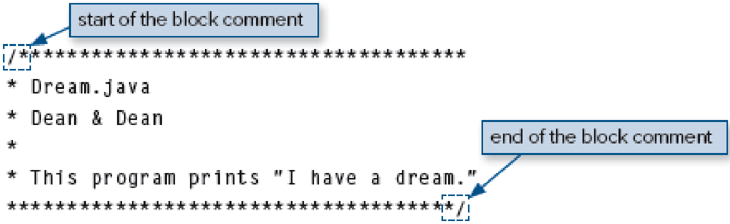
If your comment text is too long to fit on one line, you can use multiple one-line comments, but it's a bit of a pain to retype the //'s for every line. As an alternative, you can use a block comment. Block comments start with an opening /* and end with a closing */. Here's an example:

```
/*
The following code displays the androids in a high-speed chase,
wreaking havoc on nearby vehicles.
*/
```

The compiler ignores everything between the first slash and the last slash.

Prologue

We use the term *prologue* to describe a special example of a block comment. It provides information about the program so that a programmer can glance at it quickly and get an idea of what the program is all about. You should put a prologue at the top of every one of your programs. To make it stand out, it's common to enclose the prologue in a box of asterisks. Here's the Dream program's prologue:



start of the block comment

```
/************************************
* Dream.java
* Dean & Dean
*
* This program prints "I have a dream."
************************************/
```

end of the block comment

Note that the opening /* and the closing */ blend in with the other asterisks. That's OK. The compiler still recognizes the /* and */ as the start and end points of the block comment.

Include these items in your program's prologue section:

- a line of *'s
- filename
- programmer's name
- a line with a single * at its left
- program description
- a line of *'s

Readability and Blank Lines

We say that a program is *readable* if a programmer can easily understand what the program does. Comments are one way to improve a program's readability. Another way to improve a program's readability is to use blank lines. How are blank lines helpful? Isn't it easier to understand several short, simple recipes rather than a single long, complicated recipe? Likewise, it's easier to understand small chunks of code rather than one large chunk of code. Using blank lines allows you to split up large chunks of code into smaller chunks of code. In a prologue, we insert a blank line to separate the filename-author section from the description section. Also, we insert a blank line below the prologue to separate it from the rest of the program.

By the way, computers don’t care about readability; they just care about whether a program works. More specifically, computers skip all comments, blank lines, and contiguous space characters. Because computers don’t care about readability, your computer would be perfectly happy to compile and execute this Dream program:

```
public class Dream{public static void
main(String[]args){System.out.println("I have a dream!");}}
```

⚠ But a person trying to read the program would probably be annoyed because of the program’s poor readability.

3.4 The Class Heading

So far, we've focused on code that the computer ignores—comments. Now let's talk about code that the computer pays attention to. Here's the first noncomment line in the Dream program:

```
public class Dream
```

That line is called a *class heading* because it's the heading for the definition of the program's *class*. What's a class? For now, think of a class simply as a container for your program's code.

Let's examine the three words in the class heading. First, the last word—Dream. Dream is the name of the class. The compiler allows the programmer to choose any name for the class, but in the interest of making your code readable, you should choose a word or words that describe the program. Because the Dream program prints "I have a dream," Dream is a reasonable class name.

The first two words in the class heading, public and class, are *reserved words.* Reserved words, also called *keywords,*3 are words that are defined by the Java language for a particular purpose. They cannot be redefined by a programmer to mean something else. That means programmers cannot use reserved words when choosing names in their programs. For example, we were able to choose Dream for the class name because *Dream* is not a reserved word. We would not have been allowed to choose public or class for the class name.

So what are the meanings of the public and class reserved words? The word class is a marker that signifies the beginning of the class. For now, with our simple one-class programs, the word class also signifies the beginning of the program.

The word public is an *access*

modifier—it modifies the class's permissions so that the class is accessible by the "public." Making the class publicly accessible is crucial so that when a user attempts to run it, the user's run command will be able to find it.

There are certain coding conventions that most programmers follow. We list such conventions in our "Java Coding-Style Conventions" appendix. Throughout the book, when we refer to "standard coding conventions," we're referring to the coding conventions found in the appendix. Standard coding conventions dictate that class names start with an uppercase first letter; thus, the D in the Dream class name is uppercase. Java is *case-sensitive,* which means that the Java compiler distinguishes between lowercase and uppercase letters. Because Java is case-sensitive, the filename should also start with an uppercase first letter.

3.5 The main Method's Heading

We've talked about the class heading. Now it's time to talk about the heading that goes below the class heading—the main method heading. In starting a program, the computer looks for a main method heading, and execution begins with the first statement after the main method heading. The main method heading must have this form:

```
public static void main(String[] args)
```

Let's start our analysis of the main method heading by explaining the word main itself. So far, all you know about main is that in starting a program, the computer looks for it. But main is more than that; it's a Java *method*. A Java method is similar to a mathematical function. A mathematical function takes arguments, performs a calculation, and returns an answer. For example, the sin(x) mathematical function takes the x argument, calculates the sine of the given x angle, and returns the calculated sine of x. Likewise, a Java method may take arguments, will perform a calculation, and may return an answer.

The rest of the main heading contains quite a few mysterious words whose explanations may be confusing at this point. In later chapters, when you’re better prepared, we’ll explain the words in detail. For now, it’s OK to treat the main method heading as a line of text that you simply copy and paste under the class heading. We realize that some of you may be uncomfortable with that. For you folks, the rest of this section explains main method heading details.

Explanation of main Method Heading Details

Page 70

We'll now explain the three reserved words at the left of the main method heading—public static void. As previously mentioned, the word public is an access modifier—it grants permissions so that main is accessible by the "public." Because main is the starting point for all Java programs, it must be publicly accessible.

While public specifies who can access the main method (everyone), the word static specifies how to access the main method. With a non-static method, you must do some extra work prior to accessing it.[4] On the other hand, a static method can be accessed immediately, without doing the extra work. Because main is the starting point for all Java programs, it must be immediately accessible, and therefore it requires the word static.

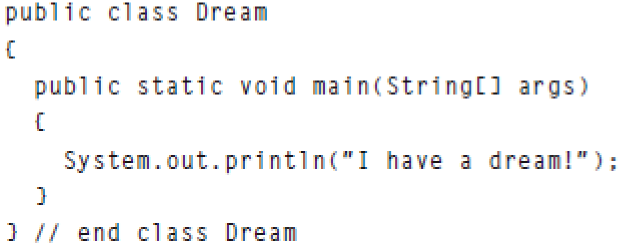
Now for the third reserved word in the main heading—void. Remember that a method is like a mathematical function—it calculates something and returns the calculated value. Well actually, a Java method sometimes returns a value and sometimes returns nothing. void indicates that a method returns nothing. Because the main method returns nothing, we use void in the main method's heading.

Now for the (String[] args) portion of the main heading. Remember that a mathematical function takes arguments. Likewise, the main method takes arguments.[5] Those arguments are represented by the word args. In Java, if you ever have an argument, you need to tell the computer what type of value the argument can hold. In this case, the argument's type is defined to be String[], which tells the computer that the args argument can hold an array of strings. The square brackets,

[], indicate an array. An *array* is a structure that holds a collection of elements of the same type. In this case String[] is an array that holds a collection of strings. A *string* is a sequence of characters. You'll learn more about strings later in this chapter in Section 3.22, and you'll learn about arrays in Chapter 9.

3.6 Braces

In the Dream program, we inserted opening braces, {, below the class heading and below the main heading, and we inserted closing braces, }, at the bottom of the program. Braces identify groupings for humans and for the computer. They must come in pairs —whenever you have an opening brace, you'll need an associated closing brace. In the Dream program, the top and bottom braces group the contents of the entire class, and the interior braces group the contents of the main method. For readability's sake, you should put an opening brace on a line by itself in the same column as the first character of the previous line. Look at the following code fragment and note how the opening braces are positioned correctly.



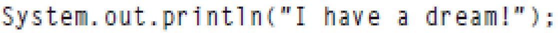
```
public class Dream
{
  public static void main(String[] args)
  {
    System.out.println("I have a dream!");
  }
} // end class Dream
```

The first brace is positioned immediately below the first character in the class heading, and the second brace is positioned immediately below the first character in the main heading. For readability's sake, you should put a closing brace on a line by itself in the same column as its partner opening brace. Look at the above code fragment and note how the closing braces are positioned correctly.

In pseudocode, remember how you indented statements that were logically inside something else? You should do the same thing in Java. You can rely on the braces to remind you when something is inside something else. In the Dream program, the Dream class's braces surround the main method, so you should indent the entire main method. Likewise, the main method's braces surround the System.out.println statement, so you should indent that statement further. Speaking of the System.out.println statement. . . .

3.7 System.out.println

In the Dream program, the main method contains this one statement:

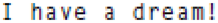


```
System.out.println("I have a dream!");
```

The System.out.println statement tells the computer to print something. The word System refers to the computer.

System.out refers to the output part of the computer system—the computer's monitor. The word println (pronounced "print line") refers to the Java println method that's in charge of printing a message to the computer screen. The above statement would normally be referred to as a println method call. You *call* a method when you want to execute it.

The parentheses after println contain the message that is to be printed. The above statement prints this message on a computer screen:



```
I have a dream!
```

Note the double quotes in System.out.println("I have a dream!"); To print a group of characters (e.g., I, space, h, a, v, e, . . .), you need to group them together. As you learned in Chapter 2, the double quotes are in charge of grouping together characters to form a string literal.

Note the semicolon at the end of

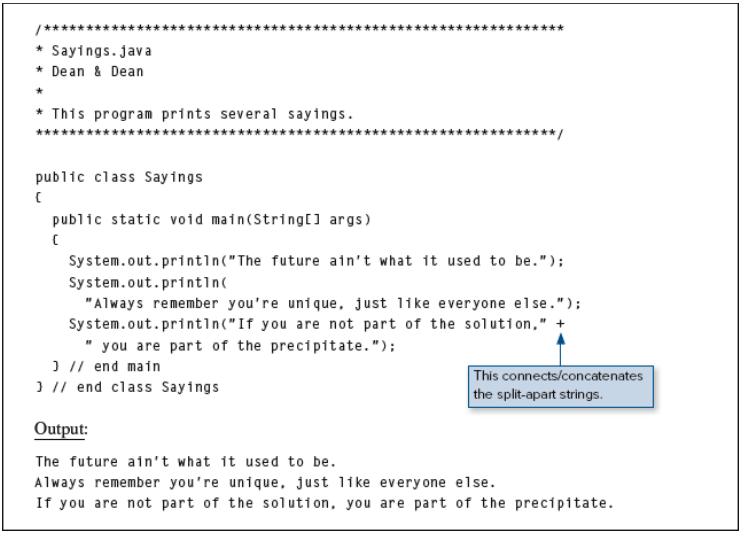
System.out.println("I have a dream!"); A semicolon in the Java language is like a period in natural language. It indicates the end of a statement. You’ll need to put a semicolon at the end of every System.out.println statement.

You’ll be calling the System.out.println method a lot, so you might want to try to memorize its wording. To help with your memorization, think of it as an acronym—“Sop” for System, out, and println. Don’t forget that the *S* is uppercase and the rest of the command is lowercase.

The System.out.println method prints a message and then moves to the beginning of the next line. That means that if there is another System.out.println method call, it starts its printing on the next line. The upcoming example illustrates what we’re talking about.

An Example

In our Dream program, we print just one short line—“I have a dream!” In our next example, we print multiple lines of varying lengths. See Figure 3.2’s Sayings program and its associated output. Note how each of the three println method calls produces a separate line of output. Note how the second println method call is too long to fit on one line, so we split it just to the right of the left parenthesis. The third println method call is longer than the second println method call, and as such, it could not fit on two lines if it were split after the left parenthesis. In other words, this does not work:



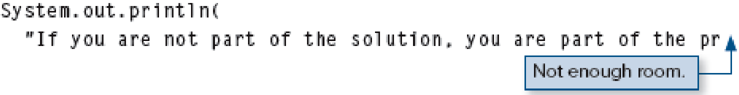
```
/*******************************************************************
* Sayings.java
* Dean & Dean
*
* This program prints several sayings.
*******************************************************************/

public class Sayings
{
  public static void main(String[] args)
  {
    System.out.println("The future ain't what it used to be.");
    System.out.println(
      "Always remember you're unique, just like everyone else.");
    System.out.println("If you are not part of the solution," +
      " you are part of the precipitate.");
  } // end main
} // end class Sayings
```

This connects/concatenates the split-apart strings.

Output:

```
The future ain't what it used to be.
Always remember you're unique, just like everyone else.
If you are not part of the solution, you are part of the precipitate.
```

Figure 3.2 Sayings program and its associated output



```
System.out.println(
  "If you are not part of the solution, you are part of the pr
```

Not enough room.

Thus, we split the third println method call in the middle of the string that is to be printed. To split a string literal, you need to put opening and closing quotes around each of the two split-apart substrings, and you need to insert a + between the substrings. See the quotes and the + in Figure 3.2's third println method call.

3.8 Compilation and Execution

Up to this point in the chapter, you've been exposed only to the theory behind Java code (the theory behind the Dream program's code and the theory behind the Sayings program's code). To gain a more complete appreciation for code, you need to enter it on a computer, compile it, and run it. After all, learning how to program requires lots of hands-on practice. It's a "contact sport"! We've provided several tutorials on the book's website that step you through the compilation and execution of a few simple Java programs. We recommend that you now take the time to work your way through one or more of those tutorials. The rest of this section covers some basic concepts related to compilation and execution. Be aware that we cover these concepts plus additional details in the tutorials.

After entering a program's source code on a computer, save it in a file whose name is comprised of the class name plus a .java extension. For example, because the Dream program's class name is Dream, its source code filename must be Dream.java.

After saving a program's source code in an appropriately named file, create Java bytecode6 by submitting the source code file to a Java compiler. In compiling the source code, the compiler generates a bytecode program file whose name is comprised of the class name plus a .class extension. For example, because the Dream program's class name is Dream, its bytecode filename will be Dream.class.

The next step after creating the bytecode program file is to run it. To run a Java program, submit the bytecode program file to the Java virtual machine (JVM).

3.9 Identifiers

So far in this chapter, you've learned Java by looking at code. Eventually, you'll need to learn it by writing your own code. When you do so, you'll need to pick out names for your program components. Java has certain rules for naming your program components. We'll look at those rules now.

An *identifier* is the technical term for a program component's name—the name of a class, the name of a method, and so on. In our Dream program, Dream was the identifier for the class name, and main was the identifier for the method name.

Identifiers must consist entirely of letters, digits, dollar signs ($), and/or underscore (_) characters. The first character must not be a digit. If an identifier does not follow these rules, your program won't compile.

Coding-convention rules are narrower than compiler rules when it comes to identifiers. Coding conventions suggest that you limit identifiers to just letters and digits. Do not use dollar signs, and (except for named constants—to be described later) do not use underscores. They also suggest that you use lowercase for all your identifier letters except:

- Start class names with an uppercase letter. For example, our Dream class starts with an uppercase D.
- Run together the words in a multiple-word identifier, using an uppercase letter for the first letter in the second word, third word, and so on. For example, if a method prints a favorite color, an appropriate method name would be printFavoriteColor.

Perhaps the most important coding-convention identifier rule is the one that says identifiers must be descriptive. Returning to the example of a method that prints a favorite color, printFavoriteColor is plenty descriptive. But how about favColor? Nope, not good enough. Some programmers like to use abbreviations (like “fav”) in their identifiers. That works OK sometimes, but not all that often. We recommend staying away from abbreviations unless they’re standard. Using complete and meaningful words in identifiers promotes self documentation. A program is *self-documenting* if the code itself explains the meaning, without needing a manual or lots of comments.

If you break a coding-conventions rule, it won’t affect your program’s ability to compile, but it will detract from your program’s readability. Suppose you have a sngs method that prints a list of the week’s top 40 songs. Even though sngs might work, you should rename it to something like printTop40Songs to improve your program’s readability.

3.10 Variables

To this point, our programs haven’t done a whole lot; they’ve just printed a message. If you want to do more than that, you’ll need to be able to store values in variables. A Java variable can hold only one type of value. For example, an integer variable can hold only integers and a string variable can hold only strings.

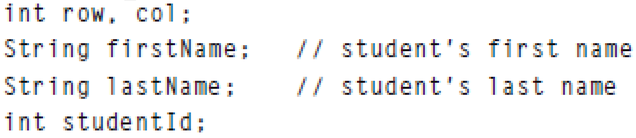
Variable Declarations

How does the computer know which type of data a particular variable can hold? Before a variable is used, its type must be declared in a *declaration* *statement.*

Declaration statement syntax:

type list-of-variables-separated-by-commas;

Example declarations:



```
int row, col;
String firstName;   // student's first name
String lastName;    // student's last name
int studentId;
```

In each declaration statement, the word on the left specifies the type for the variable or variables on the right. For example, in the first declaration statement, int is the type for the row and col variables. Having an int type means that the row and col variables can hold only integers (int stands for integer). In the second declaration statement, String is the type for the firstName variable. Having a String type means that the firstName variable can hold only strings.

Have you noticed that we sometimes spell *string* with an uppercase *S* and we sometimes spell it with a lowercase *s*? When we use "string" in the general sense, to refer to a sequence of characters, we use a lowercase *s*. In Java, String is a data type that happens to be a class name also. As you now know, coding conventions dictate that class names begin with an uppercase letter. Thus, the String class/data type begins with an uppercase S. So when we refer to String as a data type, in code and in conversational text, we use an uppercase S.

When you declare a variable or variables, don't forget to put a semicolon at the end of each declaration statement. When you declare more than one variable with one declaration statement, don't forget to separate the variables with commas.

Style Issues

The compiler will accept a variable declaration anywhere in a block of code, as long as it's above where the variable is used. However, in the interest of readability, you should normally put your declarations at the top of the main method. That makes them easy to find.

Although it may waste some space, you should normally declare only one variable per declaration statement. That way, you'll be able to provide a comment for each variable (and you should normally provide a comment for each variable).

We do make exceptions to these recommendations. Note how these row and col variables are declared together with one declaration statement:

```
int row, col;
```

That's acceptable because they are intimately related. Note that the row and col variables are declared without a comment. That's acceptable because row and col are standard names that all programmers should understand. It would be overkill to include a comment like this:

```
int row, col;      // row and col hold row and column index numbers
```

Note how this studentId variable is declared without a comment:

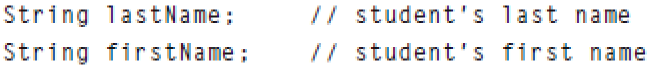
```
int studentId;
```

That’s acceptable because the studentId name is so completely descriptive that everyone should be able to understand it. It would be overkill to include a comment like this:

```
String studentId;  // a student's ID value
```

Variable names are identifiers. Thus, when you name your variables, you should follow the identifier rules covered earlier. The studentId variable is well named—it uses all lowercase letters except for the first letter in its second word, Id.

One final recommendation for your variable declarations: Try to align your comments so that they all begin in the same column. For example, note how the //'s are in the same column here:



```
String lastName;     // student's last name
String firstName;    // student's first name
```

3.11 Assignment Statements

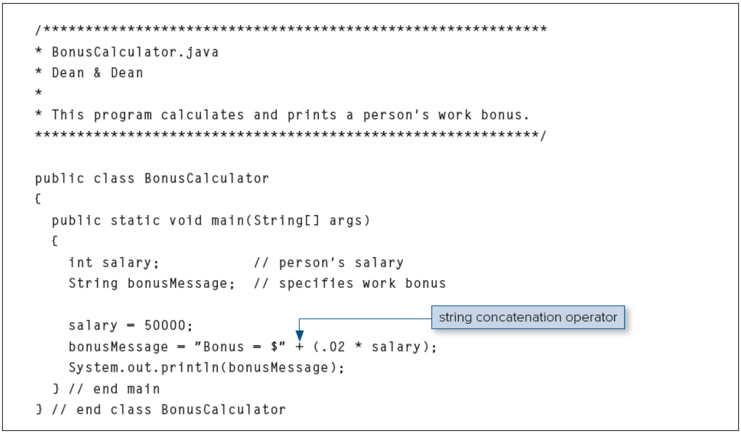
You now know how to declare a variable in Java. After declaring a variable, you'll want to use it, and the first step in using a variable is to put a value inside it. We'll now consider the assignment statement, which allows you to assign/put a value into a variable.

Java Assignment Statements

Java uses the single equals sign (=) for assignment statements. See Figure 3.3's BonusCalculator program. In particular, note the salary = 50000; line. That's an example of a Java assignment statement. It assigns the value 50000 to the variable salary.

In the BonusCalculator program, note the blank line below the declaration statements. In accordance with the principles of good style, you should insert blank lines between logical chunks of code. A group of declaration statements is usually considered to be a logical chunk of code, so you should normally insert a blank line below your bottom declaration statement.

Let's analyze the program's bonusMessage assignment statement. Note the * operator. The * operator performs multiplication. Note the + operator. If a + operator appears between a string and something else (e.g., a number or another string), then the + operator performs *string concatenation*. That means that the JVM appends the item at the right of the + to the item at the left of the +, forming a new string. In our example, the mathematical expression, .02 * salary, is evaluated first since it's inside parentheses. The JVM then appends the result, 1000, to "Bonus = $", forming the new string "Bonus = $1000".



```
/**************************************************************
* BonusCalculator.java
* Dean & Dean
*
* This program calculates and prints a person's work bonus.
**************************************************************/

public class BonusCalculator
{
  public static void main(String[] args)
  {
    int salary;           // person's salary
    String bonusMessage;  // specifies work bonus

    salary = 50000;
    bonusMessage = "Bonus = $" + (.02 * salary);
    System.out.println(bonusMessage);
  } // end main
} // end class BonusCalculator
```

string concatenation operator

Figure 3.3 BonusCalculator program

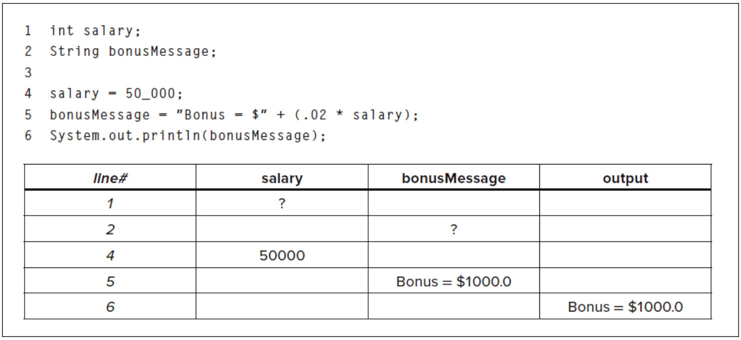
In the bonusMessage assignment statement, note the parentheses around .02 * salary. Although the parentheses are not required by the compiler, we prefer to include them here because they improve the code's readability. They improve readability by making it clear that the math operation (.02 × salary) is separate from the string concatenation operation. Use of discretionary parentheses to enhance clarity is an art. Sometimes it’s helpful, but don’t get carried away. If you use parentheses too often, your code can look cluttered.

In the salary assignment statement, note the 50000. You might be tempted to insert a comma in 50000 to make it read better; that is, you might be tempted to enter 50,000. If you do insert the comma, your program will not compile successfully. On the other hand, it is legal to use underscores to separate groups of digits. So in the BonusCalculator program, we could have used 50_000 instead of 50000. This underscore feature (which we’ll describe in more detail in Section 3.14) can come in handy for separating groups of digits in a phone number or a social security number. But be aware that you can use underscores only for numbers that are part of your source code. If you enter a number as input, you must use all digits—no underscores or commas!

Tracing

As part of a program's presentation, we'll sometimes ask you to trace the program. Tracing forces you to understand program details thoroughly. And understanding program details thoroughly is important for writing good programs. To set up a trace, provide a column heading for each variable and for output. Then execute each statement, starting with the first statement in main. For declaration statements, write a ? in the declared variable's column, indicating that the variable exists, but it doesn't have a value yet. For assignment statements, write the assigned value in the variable's column. For a print statement, write the printed value in the output column.[7]

For your first Java trace, we'll make things easy. Rather than asking you to do a trace on your own, we just ask you to study the completed trace in Figure 3.4. But please do study it. Make sure you understand how all the column values get filled in.[8]



```
1  int salary;
2  String bonusMessage;
3
4  salary = 50_000;
5  bonusMessage = "Bonus = $" + (.02 * salary);
6  System.out.println(bonusMessage);
```

line#	salary	bonusMessage	output
1	?		
2		?	
4	50000		
5		Bonus = $1000.0	
6			Bonus = $1000.0

Figure 3.4 Calculating a bonus—code fragment and its associated trace

3.12 Initialization Statements

A declaration statement specifies a

data type for a particular variable. An assignment statement puts a value into a variable. An initialization statement is a combination of the declaration and assignment statements—it specifies a data type for a variable, and it puts a value into that variable.

The Java language is *strongly typed,* meaning that all variable types are fixed. Once a variable is declared, it cannot be redeclared. Therefore, you can have only one declaration statement for a particular variable. Likewise, since an initialization statement is a specialized form of a declaration statement, you can have only one initialization statement for a particular variable.

Here's the syntax for an initialization statement:

```
type variable = value;
```

And here are some initialization examples:

```
String name = "John Doe"; // student's name
int creditHours = 0;      // student's total credit hours
```

The name variable is declared to be a String type, and it’s given the initial value of “John Doe.”[9] The creditHours variable is declared to be an int, and it’s given the initial value of 0.

Page 78

Here’s an alternative way to do the same thing using declaration and assignment statements (instead of using initialization statements):

```
String name;         // student's name
int creditHours;     // student's total credit hours

name = "John Doe";
creditHours = 0;
```

It’s OK to use either technique—initialization or declaration/assignment. You’ll see it done both ways in the real world. Initialization has the benefit of compactness. Declaration/assignment has the benefit of leaving more room in the declaration for a comment.

3.13 Numeric Data Types

—int, long, float, double

Integers

We've already mentioned one Java numeric data type—int. We'll now discuss numeric types in more detail. Variables that hold whole numbers (e.g., 1000, -22) should normally be declared with the int data type or the long data type. A whole number is a number with no decimal point and no fractional component.

An int uses 32 bits of memory. A long uses 64 bits of memory (twice as many bits as an int). The range of values that can be stored in an int variable is approximately -2 billion to $+2$ billion. The range of values that can be stored in a long variable is approximately -9×10^{18} to $+9 \times 10^{18}$. Here's an example that declares studentId to be an int variable and satelliteDistanceTraveled to be a long variable:

```
int studentId;
long satelliteDistanceTraveled;
```

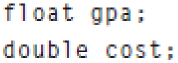
If you attempt to store a really big number (a number over 2 billion) in an int variable, you’ll get an “integer number too large” error when you compile your program. So to be safe, why shouldn’t you just always declare your integer variables as type long rather than type int? An int takes up less storage in memory. And using less storage means your computer will run faster because there’s more free space. So in the interest of speed/efficiency, use an int rather than a long for a variable that holds values less than 2 billion.[10] If you’re not sure whether a variable will hold values greater than 2 billion, play it safe and use a long. If you want the greatest possible precision in financial calculations, convert everything to cents, and use long variables to hold all values.

Floating-Point Numbers

Most mathematics books refer to numbers that contain a decimal point (like 66. and -1234.5) as real numbers. In Java, such numbers are called *floating-point* numbers. Why? Because a floating-point number can be written with different forms by shifting (floating) its decimal point. For example, the number -1234.5 can be written equivalently as -1.2345×10^3. See how the decimal point has "floated" to the left in the second version of the number?

There are two types for floating-point numbers—float and double. A float uses 32 bits of memory. A double uses 64 bits of memory. A double is called a "double" because it uses twice as many bits as a float.

Here's an example that declares gpa as a float variable and cost as a double variable:



```
float gpa;
double cost;
```

The double data type is used much more often than the float data type. You should normally declare your floating-point variables to be double rather than float because (1) double variables can hold a wider range of numbers[11] and (2) double variables can store numbers with greater precision. Greater precision means more significant digits. You can rely on 15 significant digits for a double variable but only 6 significant digits for a float variable.

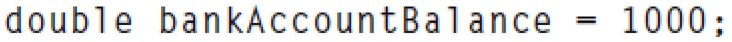
Six significant digits may seem like a lot, but for many cases, six significant digits are not enough. With only six significant digits, accuracy errors can creep into float-based programs whenever there’s a mathematical operation (addition, multiplication, etc.). If such a program performs a significant number of mathematical operations, then the accuracy errors become nontrivial. So as a general rule, use double rather than float for programs that perform a significant number of floating-point mathematical operations. And because accuracy is particularly important with money, scientific measurements, and engineering measurements, use double rather than float for calculations that involve those items.

The 15 significant digits in a double variable should be sufficient for all your floating-point programming needs, but if you come across a situation where you need a floating-point variable with more than 15 significant digits, you can declare your variable using the word BigDecimal rather than the word -double. BigDecimal variables handle an unlimited number of significant digits. For a thorough understanding of BigDecimal, you'll need to learn a few advanced concepts that are better off postponed. Nonetheless, if you're champing at the bit,[12] feel free to jump ahead to Chapter 12 for a sneak preview. Chapter 12 describes not only BigDecimal, but also BigInteger. BigInteger variables store integers with an unlimited number of significant digits.

Assignments Between Different Types

You've learned about assigning integer values into integer variables and floating-point values into floating-point variables, but you haven't learned about assignments where the types are different.

Assigning an integer value into a floating-point variable works just fine. Note this example:



```
double bankAccountBalance = 1000;
```

Assigning an integer value into a floating-point variable is like putting a small item into a large box. The int type goes up to approximately 2 billion. It’s easy to fit 2 billion into a double “box” because a double goes all the way up to 1.8×10^{308}.

On the other hand, assigning a floating-point value into an integer variable is like putting a large item into a small box. You can’t do it. For example, this generates an error: ⚠

```
int temperature = 26.7;
```

Because 26.7 is a floating-point value, it cannot be assigned to the int variable, temperature. That should make sense when you realize that it’s impossible to store .7, the fractional portion of 26.7, in an int. After all, int variables don’t store fractions; they store only whole numbers.

This statement also generates an error:

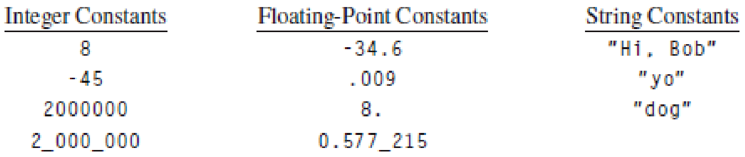
```
int count = 0.0;
```

The rule says that it's illegal to assign a floating-point value to an integer variable. 0.0 is a floating-point value. It doesn't matter that the fractional portion of 0.0 is insignificant (it's .0); 0.0 is still a floating-point value, and it's always illegal to assign a floating-point value into an integer variable. That type of error is known as a *compile-time error* or *compilation error* because the error is identified by the compiler during the compilation process.

Later in the book, we provide additional details about integer and floating-point data types. You don't need those details now, but if you can't wait, you can find the details in Chapter 12, Section 12.2.

3.14 Constants

We've used numeric and string values in our examples, but we haven't given you the formal name for them. Numeric and string values are called *constants*. They're called constants because their values are fixed—they don't change. Here are some examples:



Integer Constants	Floating-Point Constants	String Constants
`8`	`-34.6`	`"Hi, Bob"`
`-45`	`.009`	`"yo"`
`2000000`	`8.`	`"dog"`
`2_000_000`	`0.577_215`	

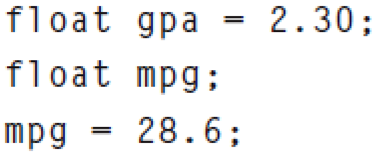
In the bottom line, note the underscores in the integer and floating-point constants. For large numbers, underscores can help to improve readability. For example, it's easy to identify the bottom integer as 2 million. Likewise, it's easy to see that the bottom floating-point number contains 6 digits of precision. In evaluating an underscored number, the JVM discards the underscores and just pays attention to the number itself. Thus, in Java, 2_000_000 equals 2_00_00_00. It's legal to place underscores between any two digits within a number (thus precluding underscores at the number's left side, at the number's right side, or next to the decimal point). But because most countries group digits by the thousands, you should normally use underscores only between every third digit. Although many countries use a comma, point, or space for the digit group separator, those symbols won't work in Java. For example, if you try to use 2,000,000 (for 2 million) in a Java program, you'll get a compilation error.

For a constant to be a floating-point constant, it must contain a decimal point, but numbers to the right of the decimal point are optional. Thus, 8. and 8.0 represent the same floating-point constant.

What is the default type for integer constants—int or long? You'd probably guess int because *integer* sounds like int. And that guess is correct—the default type for an integer constant is int. So the above integer examples (8, −45, and 2000000) are all int constants.

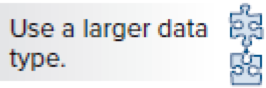
⚠ What is the default type for floating-point constants—float or double? Although you might be tempted to say float because “floating point” sounds like “float”, the correct answer is double. Because this is so easily forgotten, let us repeat: The default type for floating-point constants is double.

Try to identify the compile-time errors in this code fragment:



```
float gpa = 2.30;
float mpg;
mpg = 28.6;
```

The 2.30 and 28.6 constants both default to type double, which uses 64 bits. The 64 bits can’t squeeze into the 32-bit gpa and mpg variables so this code generates “possible loss of precision” error messages.



There are two possible solutions for these types of errors. The easiest solution is to use double variables instead of float variables all the time. Here’s another solution: Explicitly force the floating-point constants to be float by using an f or F suffix, like this:

```
float gpa = 2.30f;
float mpg;
mpg = 28.6F;
```

Two Categories of Constants

Constants can be split into two categories—hardcoded constants and named constants. The constants we’ve covered so far can be referred to as hardcoded constants. A *hardcoded constant* is an explicitly specified value. Hardcoded constants are also called *literals*. “Literal” is a good, descriptive term because literals refer to items that are interpreted literally; for example, 5 means 5, “hello” means “hello.” In the following statement, the forward slash (/) is the division operator, and 299_792_458.0 is a hardcoded constant:

```
propagationDelay = distance / 299_792_458.0;
```

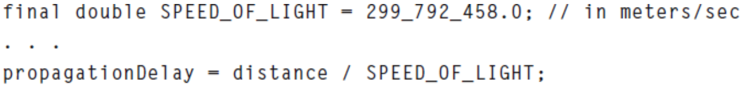
Assume that this code fragment is part of a program that calculates delays in messages carried through space. What’s the meaning behind the value 299_792_458.0? Not very obvious, eh? Read on.

In space, message signals travel at the speed of light. Because time = distance / velocity, the time it takes a message signal to travel from a satellite equals the satellite’s distance divided by the speed of light. Thus, in the code fragment, the number 299_792_458.0 represents the speed of light.

The above code fragment is somewhat confusing. The meaning behind the hardcoded constant 299_792_458.0 may be clear to science techies, but it isn't very clear to the rest of us. For a better solution, use a named constant.

Named Constants

A *named constant* is a constant that has a name associated with it. For example, in this code fragment, SPEED_OF_LIGHT is a named constant:



```
final double SPEED_OF_LIGHT = 299_792_458.0; // in meters/sec
. . .
propagationDelay = distance / SPEED_OF_LIGHT;
```

As you should be able to discern from this code fragment, a named constant is really a variable. Now there's an oxymoron—a constant is a variable. Note how SPEED_OF_LIGHT is declared to be a double variable, and it's initialized to the value 299_792_458.0. How is the SPEED_OF_LIGHT initialization different from initializations that you've seen in the past? The word final appears at the left. Also, notice that we inserted optional underscores to help identify the proper number of digits.

The reserved word final is a *modifier*—it modifies SPEED_OF_LIGHT so that its value is fixed or “final.” And being fixed is the whole point of a named constant. Thus, all named constants use the final modifier. The final modifier tells the compiler to generate an error if your program ever tries to change the final variable’s value at a later time.

Standard coding conventions suggest that you capitalize all characters in a named constant and use an underscore to separate the words in a multiple-word named constant. Example: SPEED_OF_LIGHT. The rationale for the uppercase Page 82is that uppercase makes things stand out. And you want named constants to stand out because they represent special values.

Named Constants Versus

Hardcoded Constants

Not all constants should be named constants. For example, if you need to initialize a count variable to 0, use a hardcoded 0 like this:

```
int count = 0;
```

So how do you know when to use a hardcoded constant versus a named constant? Use a named constant if it makes the code easier to understand. The above count initialization is clear the way it is now. If you replace the 0 with a named constant (e.g., int count = COUNT_STARTING_VALUE), it does not improve the clarity, so stick with the hardcoded constant. On the other hand, this code is unclear:

```
propagationDelay = distance / 299_792_458.0;
```

By replacing 299_792_458.0 with a SPEED_OF_LIGHT named constant, it does improve the clarity, so switch to the named constant.

There are two main benefits of using

named constants:

1. Named constants make code more self-documenting, and therefore more understandable.

2. If a programmer ever needs to change a named constant's value, the change is easy—find the named constant initialization at the top of the method and change the initialization value. That implements the change automatically everywhere within the program. There is no danger of forgetting to change one of many occurrences of some constant value. There is consistency.

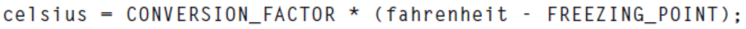
Make it easy to change.

An Example

Let's put what you've learned about constants into practice by using them within a complete program. In Figure

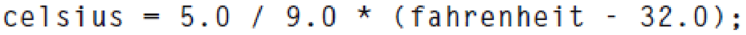
3.5’s TemperatureConverter program, we convert a Fahrenheit temperature value to a Celsius temperature value. Note the two named constant initializations at the top of the program: (1) the FREEZING_POINT named constant gets initialized to 32.0 and (2) the CONVERSION_FACTOR named constant gets initialized to 5.0 / 9.0. Usually, you’ll want to initialize each named constant to a single hardcoded constant. For example, FREEZING_POINT’s initialization value is 32.0. But be aware that it’s legal to use a constant expression for a named constant initialization value. For example, CONVERSION_FACTOR’s initialization value is 5.0 / 9.0. That expression is considered to be a constant expression because constant values are used, not variables.

In the TemperatureConverter program, this statement performs the conversion:



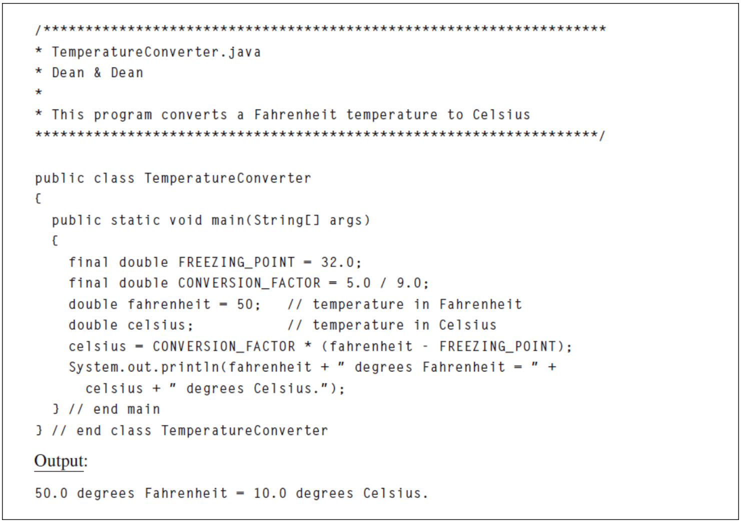
```
celsius = CONVERSION_FACTOR * (fahrenheit - FREEZING_POINT);
```

By using named constants, CONVERSION_FACTOR and FREEZING_POINT, we’re able to embed some meaning into the conversion code. Without named constants, the statement would look like this:



```
celsius = 5.0 / 9.0 * (fahrenheit - 32.0);
```

The 5.0 / 9.0 may be distracting to some readers. They may spend time wondering about the significance of the 5.0 and the 9.0. By using a CONVERSION_FACTOR named constant, we tell the reader “Don’t worry about it; it’s just a conversion factor that some scientist came up with.” If someone who is unfamiliar with the Fahrenheit scale reads the above statement, they won’t know the significance of the 32.0. Using a FREEZING_POINT named constant makes things clearer.



```
/*********************************************************************
* TemperatureConverter.java
* Dean & Dean
*
* This program converts a Fahrenheit temperature to Celsius
*********************************************************************/

public class TemperatureConverter
{
  public static void main(String[] args)
  {
    final double FREEZING_POINT = 32.0;
    final double CONVERSION_FACTOR = 5.0 / 9.0;
    double fahrenheit = 50;    // temperature in Fahrenheit
    double celsius;            // temperature in Celsius
    celsius = CONVERSION_FACTOR * (fahrenheit - FREEZING_POINT);
    System.out.println(fahrenheit + " degrees Fahrenheit = " +
      celsius + " degrees Celsius.");
  } // end main
} // end class TemperatureConverter
```

Output:

```
50.0 degrees Fahrenheit = 10.0 degrees Celsius.
```

Figure 3.5 TemperatureConverter program and its output

3.15 Arithmetic Operators

Page 83

We’ve talked about numbers for a while now—how to declare numeric variables, how to assign numbers, and how to work with numeric constants. In addition, we’ve shown a few examples of using numbers in mathematical expressions. In this section and the next two sections, we study expressions in more depth. An *expression* is a combination of operands and operators that performs a calculation. Operands are variables and constants. An operator is a symbol, like + or -, that performs an operation. In this section, we'll look at arithmetic operators for numeric data types. Later, we'll look at operators for other data types.

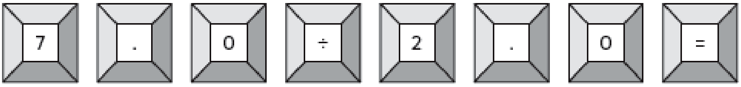
Addition, Subtraction, and Multiplication

Java's +, -, and * arithmetic operators should be familiar to you. They perform addition, subtraction, and multiplication, respectively.

Floating-Point Division

Java performs division differently depending on whether the numbers/ operands being divided are integers or whether they're floating-point numbers. Let’s first discuss floating-point division.

When the Java virtual machine (JVM) performs division on floating-point numbers, it performs “calculator division.” We call it “calculator division” because Java’s floating-point division works the same as division performed by a standard calculator. For example, if you enter this on your calculator, what is the result?



The result is 3.5. Likewise, this line of Java code prints 3.5:

```
System.out.println (7.0 / 2.0);
```

Although many calculators display a ÷ symbol on the key for division, Java uses the / character.

To explain arithmetic operators, we’ll need to evaluate lots of expressions. To simplify that discussion, we’ll use the ⇒ symbol. It means “evaluates to.” Thus, this next line says that 7.0 / 2.0 evaluates to 3.5:

7.0 / 2.0 ⇒ 3.5

This next line asks you to determine what 5 / 4. evaluates to:

5 / 4. ⇒ ?

5 is an int and 4. is a double. This is an example of a *mixed expression.* A mixed expression is an expression that contains operands with different data types. Because they contain a fractional component, double values are considered to be more complex than int values. Whenever there’s a mixed expression, the JVM temporarily *promotes* the less complex operand’s type so that it matches the more complex operand’s type, and then the JVM applies the operator. In the 5 / 4. expression, the JVM promotes 5 to a double and then performs floating-

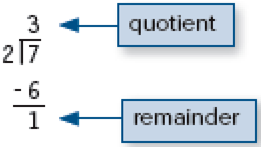
point division on the two floating-point values. The expression evaluates to 1.25.

Integer Division

When the JVM performs division on integers, it performs “grade school division.” We call it grade school division because Java’s integer division works the same as the division you did by hand in grade school. Remember how you calculated two values for each division operation? You calculated a quotient and also a remainder. Likewise, Java has the ability to calculate both a quotient and a remainder when integer division is called for. But Java doesn’t calculate both values simultaneously. If Java’s / operator is used, then the quotient is calculated. If Java’s % operator is used, then the remainder is calculated. The % operator is more formally called the *modulus* operator. Note these examples:

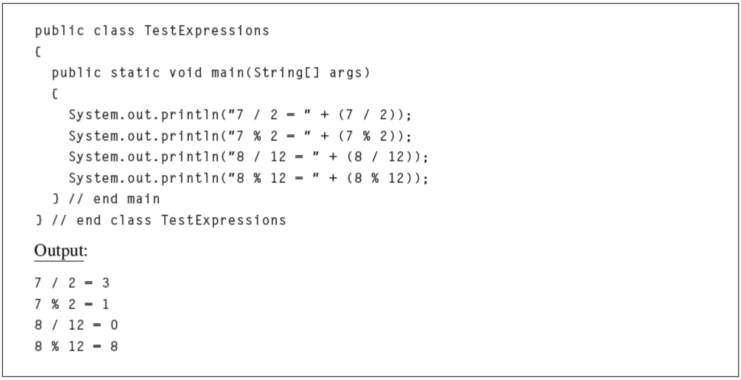
$7 / 2 \Rightarrow 3$
$7 \% 2 \Rightarrow 1$

These correspond to the equivalent grade school arithmetic notation:



Print details to see what the computer does.

We'll give you many expression evaluation problems like this. As a sanity check, we recommend that you verify at least some of the calculated results by executing the expressions on a computer. To execute the expressions, embed the expressions into print statements, embed the print statements into a test program, and run the test program. For example, to execute the above expressions, use the TestExpressions program in Figure 3.6.



```
public class TestExpressions
{
  public static void main(String[] args)
  {
    System.out.println("7 / 2 = " + (7 / 2));
    System.out.println("7 % 2 = " + (7 % 2));
    System.out.println("8 / 12 = " + (8 / 12));
    System.out.println("8 % 12 = " + (8 % 12));
  } // end main
} // end class TestExpressions
```

Output:

```
7 / 2 = 3
7 % 2 = 1
8 / 12 = 0
8 % 12 = 8
```

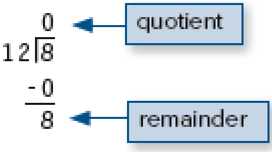
Figure 3.6 TestExpressions program and its output

Figure 3.6 also illustrates these additional examples:

8 / 12 ⇒ 0

8 % 12 ⇒ 8

Here is the corresponding grade school arithmetic notation:



Page 85

At this point, you might be thinking "Gosh, that integer division stuff sure is fun, but can I use it for something practical?" Yes, indeed! You can use it to split apart numbers into their constituent parts. For example, if you’re writing a program for a candy vending machine, each time the customer inserts a dollar, your program can use the integer division operators to split apart the customer’s change into the proper number of quarters, dimes, and nickels. You are asked to do that for one of the projects in Chapter 6. As another example, the following code fragment shows how to split apart a number that represents the current time into its constituent parts—hours and minutes:

```
int time;
System.out.print("Enter the current time as an integer (no colon): ");
time = stdIn.nextInt();
System.out.println(
  "hours = " + (time / 100) + ", minutes = " + (time % 100));
```

If the user enters 1208, then:

1208 / 100 ⇒ 12

1208 % 100 ⇒ 8

So the output would be:

```
hours = 12, minutes = 8
```

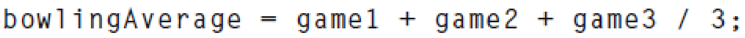
3.16 Expression Evaluation and Operator

Precedence

In the above examples, the expressions were pretty basic—they each contained only one operator—so they were fairly easy to evaluate. Expressions are sometimes fairly complicated. In this section, we discuss how to evaluate those more complicated expressions.

Average Bowling Score Example

Suppose you'd like to calculate the average bowling score for three bowling games. Would this statement work?



```
bowlingAverage = game1 + game2 + game3 / 3;
```

The code looks reasonable. But it's not good enough to rely on your sense of what looks reasonable. To be a good programmer, you need to be sure. The code you should be focusing on is the expression on the right side: game1 +

game2 + game3 / 3. More specifically, you should be asking yourself, “Which operator executes first—the left addition operator or the division operator?” To answer that question, we turn to the operator precedence table.

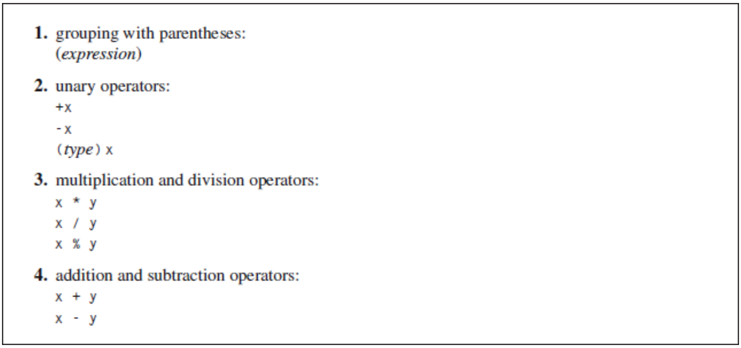
Operator Precedence Table

The key to understanding complicated expressions is to understand the operator precedence shown in Figure 3.7. Please study Figure 3.7’s operator precedence table now.

The operator precedence table might need some clarification. The groups at the top have higher precedence than the groups at the bottom. That means that if one of the top operators appears in an expression along with one of the bottom operators, then the top operator executes first. For example, if * and + both appear in the same expression, then the * operator executes before the + operator (because the * operator's group is higher in the table than the + operator's group). If parentheses appear within an expression, then the items inside the parentheses execute before the items that are outside the parentheses (because parentheses are at the very top of the table).

If an expression has two or more operators in the same group (from Figure 3.7's groups), then apply the operators from left to right. In mathematics, that's referred to as *left-to-right associativity*. In Java, that means that operators appearing on the left should be executed before operators appearing on the right. For example, because the * and / operator are in the same group, if * and / both appear in the same expression and / appears further to the left than * within that expression, division is performed before multiplication

The operators in the second-from-the-top group are *unary operators*. A unary operator is an operator that applies to just one operand. The unary + operator is cosmetic; it does nothing. The unary - operator (negation) reverses the sign of its operand. For example, if the variable x contains a 6, then -x evaluates to a negative 6. The (*type*) operator represents the cast operators. We'll get to cast operators later in this chapter.



1. grouping with parentheses:
   (*expression*)
2. unary operators:
   `+x`
   `-x`
   (*type*) x
3. multiplication and division operators:
   `x * y`
   `x / y`
   `x % y`
4. addition and subtraction operators:
   `x + y`
   `x - y`

Figure 3.7 Abbreviated operator precedence table (see Appendix 2 for the complete table) Operator groups at the top of the table have higher precedence than operator groups at the bottom of the table. All operators within a particular group have equal

precedence, and they evaluate left to right.

Average Bowling Score Example Revisited

Let's return to the average bowling score example and apply what you've learned about operator precedence. Does the following statement correctly calculate the average bowling score for three bowling games?

```
bowlingAverage = game1 + game2 + game3 / 3;
```

No. The operator precedence table says that the / operator has higher priority than the + operator, so division is performed first. After the JVM divides game3 by 3, the JVM adds game1 and game2. The correct way to calculate the average is to add the three game scores first and then divide the sum by 3. In other words, you need to force the + operators to execute first. The solution is to use parentheses like this:

```
bowlingAverage = (game1 + game2 + game3) / 3;
```

Expression Evaluation Practice

Hand calculation helps your understanding.

Let's do some expression evaluation practice problems to ensure that you really understand this operator precedence material. Given these initializations:

```
int a = 5, b = 2;
double c = 3.0;
```

What does the following expression evaluate to?

```
(c + a / b) / 10 * 5
```

Here's the solution:

1. `(c + a / b) / 10 * 5` ⇒
2. `(3.0 + 5 / 2) / 10 * 5` ⇒
3. `(3.0 + 2) / 10 * 5` ⇒
4. `5.0 / 10 * 5` ⇒
5. `0.5 * 5` ⇒
6. `2.5`

In solving expression evaluation problems, we recommend that you show each step of the evaluation process so your solution is easy to follow. In the above solution, we show each step, and we also show line numbers. There's normally no need to show line numbers, but we do it here to help with our explanation. From line 1 to line 2, we replace variables with their values. From line 2 to line 3, we evaluate the highest priority operator, the / inside the parentheses. From line 3 to line 4, we evaluate the next highest priority operator, the + inside the parentheses. Study the remaining lines on your own.

Let's do one more expression evaluation practice problem. Given these initializations:

```
int x = 5;
double y = 3.0;
```

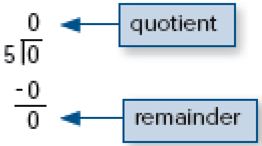
What does the following expression evaluate to?

```
(0 % x) + y + (0 / x)
```

Here's the solution:

```
(0 % x) + y + (0 / x) ⇒
(0 % 5) + 3.0 + (0 / 5) ⇒
0 + 3.0 + (0 / 5) ⇒
0 + 3.0 + 0 ⇒
3.0
```

Perhaps the trickiest part of the above solution is evaluating 0 % 5 and 0 / 5. They both evaluate to 0. This grade school arithmetic notation shows why:



3.17 More Operators: Increment, Decrement, and Compound Assignment

So far, we've covered Java math operators that correspond to operations found in math books—addition, subtraction, multiplication, and division. Java provides additional math operators that have no counterparts in math books. In this section, we'll talk about the increment, decrement, and compound assignment operators.

Increment and Decrement Operators

It’s fairly common for a computer program to count the number of times something occurs. For example, have you ever seen a web page that displays the number of “visitors”? The number of visitors is tracked by a program that counts the number of times the web page has been loaded on someone’s web browser. Since counting is such a common task for programs, there are special operators for counting. The increment operator (++) counts up by 1. The decrement operator (--) counts down by 1.

Here’s one way to increment the variable *x*:

```
x = x + 1;
```

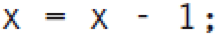
And here’s how to do it using the increment operator:

```
x++;
```

The two techniques are equivalent in

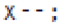
terms of their functionality. Experienced Java programmers almost always use the second form rather than the first form, and proper style suggests using the second form. So use the second form.

Here's one way to decrement the variable *x*:



```
x = x - 1;
```

And here's how to do it using the decrement operator:



```
x--;
```

Once again, you should use the second form.

Compound Assignment Operators

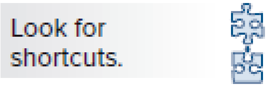
Let's now discuss five of Java's *compound assignment* operators: + =, - =, * =, / =, and % =.

The + = operator updates a variable by adding a specified value to the variable. Here's one way to increment x by 3:

```
x = x + 3;
```

And here’s how to do it using the += operator:

```
x += 3;
```


Look for shortcuts.

The two techniques are equivalent in terms of their functionality. Experienced Java programmers almost always use the shorter second form rather than the longer first form. And proper style suggests using the second form. So use the second form.

The -= operator updates a variable by subtracting a specified value from the variable. Here’s one way to decrement x by 3:

```
x = x - 3;
```

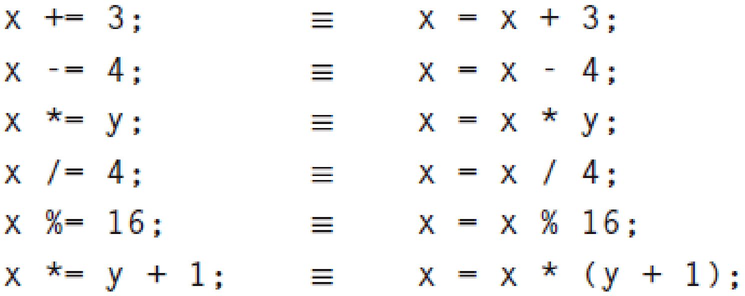
And here’s how to do it using the -= operator:

```
x -= 3;
```

Once again, you should use the second form.

The *=, /=, and %= operators

parallel the += and -= operators, so we won't bore you with detailed explanations for those remaining three operators. But we do encourage you to study the *=, /=, and %= examples shown below:



```
x += 3;          ≡    x = x + 3;
x -= 4;          ≡    x = x - 4;
x *= y;          ≡    x = x * y;
x /= 4;          ≡    x = x / 4;
x %= 16;         ≡    x = x % 16;
x *= y + 1;      ≡    x = x * (y + 1);
```

Page 90

The examples show assignment operator statements on the left and their equivalent long-form statements on the right. The ≡ symbol means "is equivalent to." It's better style to use the forms on the left rather than the forms on the right, but don't ignore the forms on the right. They show how the assignment operators work.

The bottom example is the only one in which the compound assignment operator uses an expression rather than a single value; that is, the expression to the right of the *= assignment operator is y + 1, rather than just 1. For cases like these, the compound assignment form is somewhat confusing. Therefore, for these cases, it's acceptable style-wise to use the equivalent long form rather than the compound assignment form.

Why are the +=, -=, *=, /=, and %= operators called compound assignment operators? Because they compound/combine a math operation with the assignment operation. For example, the += operator performs addition and assignment. The addition part is obvious, but what about the assignment part? The += does indeed perform assignment because the variable at the left of the += is assigned a new value.

3.18 Tracing

To make sure that you really understand the increment, decrement, and compound assignment operators, let's trace a program that contains those operators. Earlier in the chapter, we showed a trace, but the trace was for a very limited code fragment—the code fragment contained two assignment statements and that was it. In this section, we present a more complicated trace.

See the TestOperators program and associated trace table in Figure 3.8. In particular, look at the first three lines under the heading in the trace table. They contain the variables' initial values. For variables declared as part of an initialization, their initial value is the initialization value. For variables declared without an initialization, we say their initial value is *garbage* because its actual value is unknown. Use a question mark to indicate a garbage value.

Put yourself in the computer's place.

We suggest you cover up the bottom part of the trace, and try to complete the trace on your own. When you're done, compare your answer to Figure 3.8's trace table.

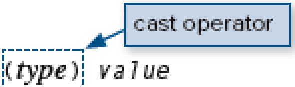
There are different modes for the increment and decrement operators—prefix mode and postfix mode. Later in the book, we explain the modes and provide details on how they work within the context of a trace. You don't need those details now, but if you can't wait, you can find the details in Chapter 12, Section 12.5.

3.19 Type Casting

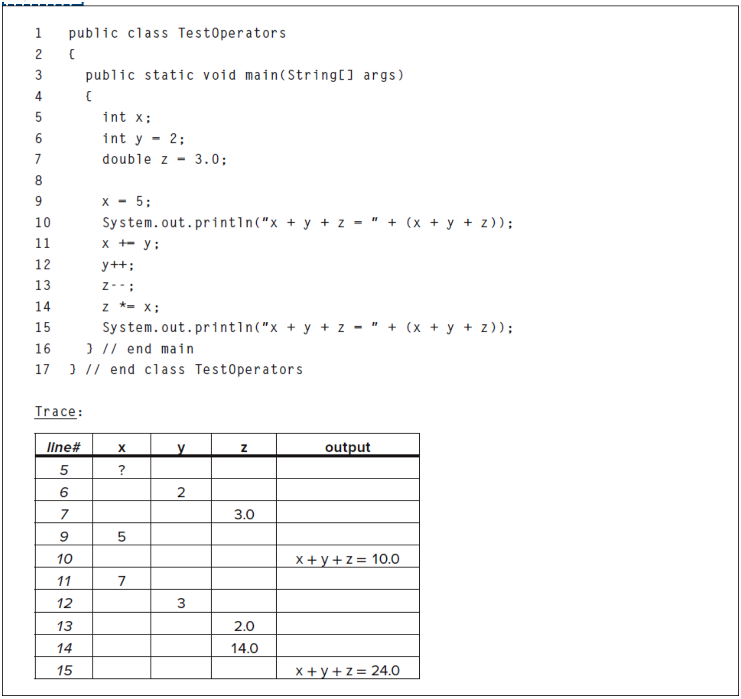
We've now described simple arithmetic operators (+, -, *, /, %), increment and decrement operators (++, --), and compound assignment operators (+=, -=, *=, /=, %=). In this section, we'll discuss yet another operator, the cast operator.

Cast Operator

In writing a program, you'll sometimes need to convert a value to a different data type. The cast operator can be used to perform that sort of conversion. Here's the syntax:



```
(type) value
```



```
1   public class TestOperators
2   {
3     public static void main(String[] args)
4     {
5       int x;
6       int y = 2;
7       double z = 3.0;
8
9       x = 5;
10      System.out.println("x + y + z = " + (x + y + z));
11      x += y;
12      y++;
13      z--;
14      z *= x;
15      System.out.println("x + y + z = " + (x + y + z));
16    } // end main
17  } // end class TestOperators
```

Trace:

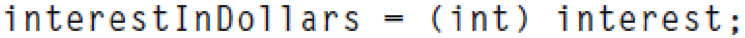
line#	x	y	z	output
5	?			
6		2		
7			3.0	
9	5			
10				x + y + z = 10.0
11	7			
12		3		
13			2.0	
14			14.0	
15				x + y + z = 24.0

Figure 3.8 TestOperators program and

its trace

As shown above, a cast operator consists of a data type inside parentheses. You should place a cast-operator at the left of the value that you'd like to convert.

Suppose you have a variable named interest that stores a bank account's interest as a double. You'd like to extract the dollars portion of the interest and store it in a variable of type int that is named interestInDollars. To do that, use the int cast operator like this:



```
interestInDollars = (int) interest;
```

The int cast operator returns the whole number portion of the casted value, truncating the fractional portion. Thus, if interest contains the value 56.96, after the assignment, interestInDollars contains the value 56. Note that the cast operation does not change the value of interest. After the assignment, interest still contains 56.96.

Use Parentheses to Cast an Expression

If you ever need to cast more than just a single value or variable, then make sure to put parentheses around the entire expression that you want to cast. Note this example:

```
double interestRate;
double balance;
int interestInDollars;
. . .
interestInDollars = (int) (balance * interestRate);
```

Parentheses are necessary here

In the interestInDollars assignment, balance * interestRate is the formula for calculating interest. This code fragment performs basically the same operation as the previous one-line code fragment. It extracts the dollars portion of the interest and stores it in an int variable named interestInDollars. The difference is that the interest this time is in the form of an expression, balance * interestRate, rather than in the form of a simple variable, interest. Because we want the cast operator to apply to the entire expression, we need to put parentheses around balance * interestRate.

⚠ In the above code fragment, what would happen if there were no parentheses around the expression, balance * interestRate? The cast would then apply only to the first thing at its right, balance, rather than the entire expression. That should make sense when you look at the operator precedence table. The operator precedence table shows that the cast operator has very high precedence. So without the parentheses, the cast operator would execute prior to the multiplication operator, and the cast would thus apply only to balance. And that leads to an incorrect calculation for interest in dollars.

Use a Floating-Point Cast to

Force Floating-Point Division

Suppose you have a variable named earnedPoints that stores a student's earned grade points for a semester's worth of classes. Suppose you have a variable named numOfClasses that stores the number of classes taken by the student. The student's grade point average (GPA) is calculated by dividing earned points by number of classes. In the following statement, earnedPoints and numOfClasses are ints and gpa is a double. Does the statement correctly calculate the student's GPA?

```
gpa = earnedPoints / numOfClasses;
```

Compare output with what you expect.

Suppose earnedPoints holds 14 and numOfClasses holds 4. You'd like gpa to get a value of 3.5 (because 14 ÷ 4 = 3.5). But alas, gpa gets a value of 3. Why? Because the / operator performs integer division on its two int operands. Integer division means the quotient is returned. The quotient of 14 ÷ 4 is 3. The solution is to force floating-point division by introducing the cast operator. Here's the corrected code:

```
gpa = (double) earnedPoints / numOfClasses;
```

After casting earnedPoints to a double, the JVM sees a mixed expression and promotes numOfClasses to a double. Then floating-point division takes place.

⚠ For this example, you should not put parentheses around the earnedPoints / numOfClasses expression. If you did so, the / operator would have higher precedence than the cast operator, and the JVM would perform division (integer division) prior to performing the cast operation.

Later in the book, we provide additional details about type conversions. You don't need those details now, but if you can’t wait, you can find the details in Chapter 12, Section 12.4.

3.20 char Type and Escape Sequences

In the past, when we’ve stored or printed text, we’ve always worked with groups of text characters (strings), not with individual characters. In this section, we’ll use the char type to work with individual characters.

char Type

If you know that you’ll need to store a single character in a variable, use a char variable. Here’s an example that declares a char variable named ch and assigns the letter *A* into it.

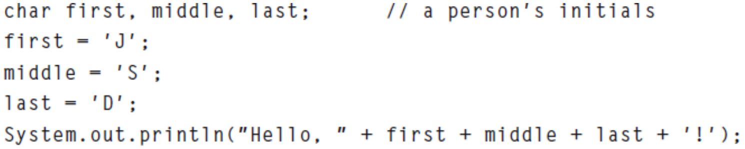
```
char ch;
ch = 'A';
```

Note the ‘A’. That’s a char literal. char literals must be surrounded by single quotes. That syntax parallels the syntax for string literals—string literals must be surrounded by double quotes.

What’s the point of having a char type? Why not just use one-character strings for all character processing? Because for applications that manipulate lots of individual characters, it’s more efficient (faster) to use char variables, which are simple, rather than string variables, which are more complex. For example, the software that allows you to view web pages has to read and process individual characters as they’re downloaded onto your computer. In processing the individual characters, it’s more efficient if they’re stored as separate char variables rather than as string variables.

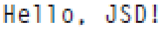
String Concatenation with char

Remember how you can use the + symbol to concatenate two strings together? You can also use the + - symbol to concatenate a char and a string. What do you think this code fragment prints?



```
char first, middle, last;     // a person's initials
first = 'J';
middle = 'S';
last = 'D';
System.out.println("Hello, " + first + middle + last + '!');
```

Here's the output:



```
Hello, JSD!
```

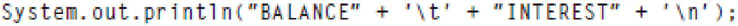
Escape Sequences

Usually, it's easy to print characters. Just stick them inside a System.out.println statement. But some characters are hard to print. We use *escape sequences* to print hard-to-print characters such as the tab character. An escape sequence is comprised of a backslash (\) and another character. See Java's most popular escape sequences in Figure 3.9.

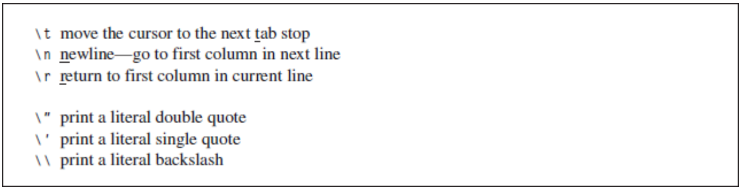
If you print the tab character (\t),

the computer screen's cursor moves to the next tab stop. The computer screen's cursor is the position on the screen where the computer prints next. If you print the newline character (\n), the computer screen's cursor moves to the beginning of the next line.

Here's an example of how you could print two column headings, BALANCE and INTEREST, separated by a tab, and followed by a blank line:



```
System.out.println("BALANCE" + '\t' + "INTEREST" + '\n');
```



`\t`	move the cursor to the next tab stop
`\n`	newline—go to first column in next line
`\r`	return to first column in current line
`\"`	print a literal double quote
`\'`	print a literal single quote
`\\`	print a literal backslash

Figure 3.9 Common escape sequences

Note that escape sequences are indeed characters, so to print the tab and newline characters, we've surrounded them with single quotes.

Normally the compiler interprets a double quote, a single quote, or a backslash as a *control character*. A control character is in charge of providing special meaning to the character(s) that follows it. The double quote control character tells the computer that the subsequent characters are part of a string literal. Likewise, the single quote control character tells the computer that the subsequent character is a char literal. The backslash control character tells the computer that the next character is to be interpreted as an escape sequence character.

But what if you'd like to print one of those three characters as is and bypass the character's control functionality? To do that, preface the control character (double quote, single quote, backslash) with a backslash. The initial backslash turns off the subsequent character's control functionality and thus allows the subsequent character to be printed as is. If that doesn't make sense, all you really have to know is this:

To print a double quote, use \".
To print a single quote, use \'.
To print a backslash, use \\.

Suppose you'd like to print this message:

```
"Hello.java" is stored in the c:\javaPgms folder.
```

Here's how to do it:

```
System.out.println('\"' + "Hello.java" + '\"' +
  " is stored in the c:" + '\\' + "javaPgms folder.");
```

Embedding an Escape Sequence Within a String

Write a print statement that generates this heading for a computer-specifications report:

```
HARD DISK SIZE        RAM SIZE ("MEMORY")
```

Specifically, your print statement should generate a tab, a HARD DISK SIZE column heading, two more tabs, a RAM SIZE ("MEMORY") column heading, and then two blank lines.

Here's one solution:

```
System.out.println('\t' + "HARD DISK SIZE" + '\t' + '\t' +
  "RAM SIZE (" + '\"' + "MEMORY" + '\"' + ")" + '\n' + '\n');
```

Look for shortcuts.

That's pretty cluttered. Fortunately, there's a better way. An escape sequence is designed to be used like any other character within a string of text, so it's perfectly acceptable to embed escape sequences within strings and omit the +'s and the single quotes. For example, here's an alternative solution for the PC specifications report heading problem where the +'s and single quotes have been removed:

```
System.out.println("\tHARD DISK SIZE\t\tRAM SIZE (\"MEMORY\")\n\n");
```

Everything is now all within just one string literal. By omitting the +'s and single quotes, the clutter is reduced and that makes everyone happy. (Exception—author John's kids love clutter and would thus abhor this second solution.)

Origin of the Word “Escape” for Escape Sequences

Why is the word “escape” used for escape sequences? The backslash forces an “escape” from the normal behavior of a specified character. For example, if t is in a print statement, the computer normally prints t. If \t is in a print statement, the computer escapes from printing t,and instead, it prints the tab character. If the double quote character (") is in a print statement, the computer normally treats it as the start or end of a string literal. If \" is in a print statement, the computer escapes from the start/end string behavior; instead the computer prints the double quote character.

Later in the book, we present relatively advanced syntax details that pertain to the char type. You don’t need those details now, but if you can’t wait, you can find the details in Chapter 12, Section 12.3.

3.21 Primitive Variables Versus Reference Variables

Throughout the chapter, we've defined and discussed various types of variables—String, int, long, float, double, and char variables. It's now time to step back and get a big-picture view of the two different categories of variables in Java—primitive variables and reference variables.

Primitive Variables

A *primitive variable* stores a single piece of data. It's helpful to think of a primitive variable's data item as being inherently indivisible. More formally, we say that it's "atomic" because, like an atom, it’s a basic “building block” and it cannot be broken apart.[13] Primitive variables are declared with a *primitive type*, and those types include:

`int, long`	(integer types)
`float, double`	(floating-point types)
`char`	(character type)

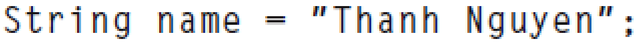
There are additional primitive types (boolean, byte, short), which we’ll get to in Chapters 4 and 12, but for most situations, these five primitive types are sufficient.

Reference Variables

Whereas a primitive variable stores a single piece of data, a *reference variable* stores a memory location that points to a collection of data. This memory location is not a literal memory address, like a street address. It’s a coded abbreviation, like a post office box number. However, for everything you can do in Java, the value in a reference variable acts exactly like a literal memory address, so we'll pretend it is one. We said a reference variable's "address" points to a collection of data. More formally, it points to an *object*. You'll learn about object details in Chapter 6, but for now, just realize that an object is a collection of related data wrapped in a protective shell. To access an object's data, you need to use a reference variable (or *reference* for short) that points to the object.

String variables are examples of reference variables. A string variable holds a memory address that points to a string object. The string object holds the data—the string's characters.

Reference variables are declared with a *reference type*. A reference type is a type that provides for the storage of a collection of data. String is a reference type, and it provides for the storage of a collection of characters. So in the following example, declaring name with a String reference type means that name points to the collection of characters T, h, a, n, h, space, N, g, u, y, e, n.

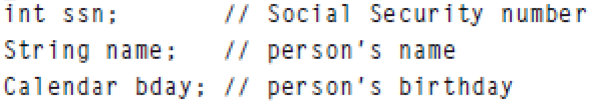


```
String name = "Thanh Nguyen";
```

String is just one reference type from among a multitude of reference types. Classes, arrays, and interfaces are all considered to be reference types. You’ll learn about arrays in Chapter 9 and interfaces in Chapter 10. You’ll learn about class details in Chapter 6, but for now, it’s good enough to know that a class is a generic description of the data in a particular type of object. For example, the String class describes the nature of the data in string objects. More specifically, the String class says that each string object can store zero or more characters and the characters are stored in a sequence.

An Example

Let’s look at an example that uses primitive variables and reference variables. In this code fragment, we declare variables that keep track of a person’s basic data:



```
int ssn;       // Social Security number
String name;   // person's name
Calendar bday; // person's birthday
```

As you can tell by the int and String data types, ssn is a primitive variable and name is a reference variable. In the third line, Calendar is a class. That tells us that bday is a reference variable. The Calendar class allows you to store date information such as year, month, and day.[14] Since bday is - declared with the Calendar class, bday is able to store year, month, and day data items.

3.22 Strings

We’ve used strings for quite a while now, but we’ve stored them and printed them and that’s it. Many -

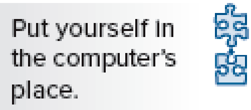
programs need to do more with strings than just store and print. For example, Microsoft Office programs (such as Word, Excel, and PowerPoint) all include text search and text replace capabilities. In this section, we describe how Java provides that sort of string-manipulation functionality in the String class.

String Concatenation

Page 97

As you know, strings are normally concatenated with the + operator. Note that strings can also be concatenated with the += compound assignment operator. In the following example, if the animal string references "dog" originally, it references "dogfish" after the statement is executed:

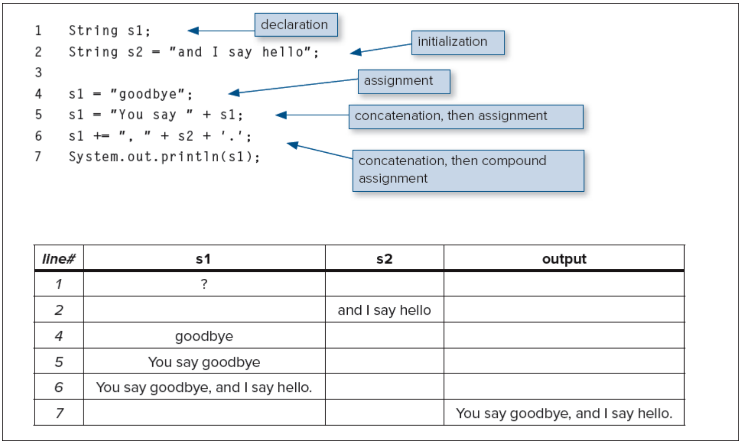
```
animal += "fish";
```



Put yourself in the computer's place.

We recommend that you now go

through a trace to make sure you thoroughly understand string concatenation. See the code fragment in Figure 3.10. Try to trace the code fragment on your own prior to looking at the solution.



```
1   String s1;
2   String s2 = "and I say hello";
3
4   s1 = "goodbye";
5   s1 = "You say " + s1;
6   s1 += ", " + s2 + '.';
7   System.out.println(s1);
```

declaration
initialization
assignment
concatenation, then assignment
concatenation, then compound assignment

line#	s1	s2	output
1	?		
2		and I say hello	
4	goodbye		
5	You say goodbye		
6	You say goodbye, and I say hello.		
7			You say goodbye, and I say hello.

Figure 3.10 Code fragment and associated trace for string concatenation illustration

String Methods

In the previous section, we defined an object to be a collection of data. An object's data is normally protected, and, as such, it can be accessed only through special channels. Normally, it can be accessed only through the object’s methods. A string object stores a collection of characters, and a string object’s characters can be accessed only through its charAt method. In the remainder of this section, we’ll describe the charAt method and three other popular string methods—length, equals, and equalsIgnoreCase. These methods, as well as many about other string methods, are defined in the String class.

Get help from the source.

If you’d like to learn more about the String class and all of its methods, you could go to https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/lang/package-summary.html and search for String.

The charAt Method

Suppose you initialize a string

variable, animal, with the value “cow”. The animal variable then points to a string object that contains three data items—the three characters ‘c’, ‘o’, and ‘w’. To retrieve a data item (i.e., a character), call the charAt method. charAt stands for character at. The charAt method returns a character at a specified position. For example, if animal calls charAt and specifies the third position, then charAt returns ‘w’ because ‘w’ is the third character in “cow”.

So how do you call the charAt method? Let us answer that question by comparing a charAt method call to a method call that you’re already comfortable with—the println method call. See Figure 3.11.

In Figure 3.11, note how the charAt method call and the println method call both use this syntax:

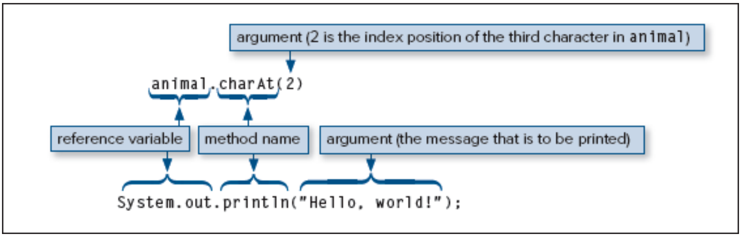
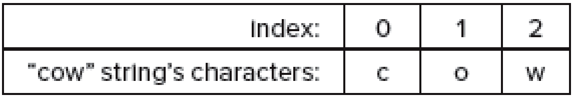

argument (2 is the index position of the third character in animal)
animal.charAt(2)
reference variable
method name
argument (the message that is to be printed)
System.out.println("Hello, world!");

Figure 3.11 Comparison of charAt method call to println method call

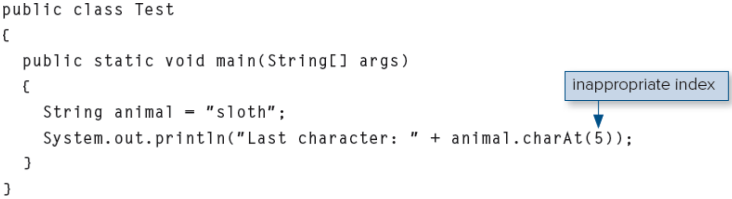
reference-variable.method-name(argument)

In the charAt call, animal is the reference variable, charAt is the method name and 2 is the argument. The argument is the tricky part. The argument specifies the *index* of the character that is to be returned. The positions of characters within a string are numbered starting with index zero, not index one. For emphasis, we say again! The positions in a string start with index zero. So if animal contains “cow,” what does animal.charAt(2) return? As the following table indicates, the ‘w’ character is at index 2, so animal.charAt(2) returns ‘w.’



index:	0	1	2
"cow" string's characters:	c	o	w

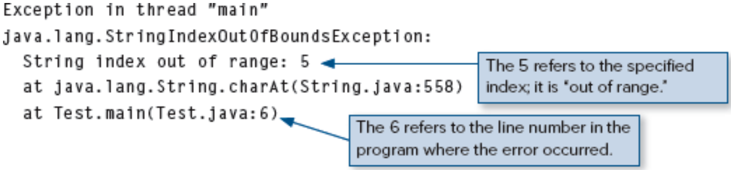
If you call charAt with an argument that's negative or that's equal to or greater than the string's length, your code will compile OK, but it won't run properly. For example, suppose you run this program:



```
public class Test
{
  public static void main(String[] args)
  {
    String animal = "sloth";
    System.out.println("Last character: " + animal.charAt(5));
  }
}
```

inappropriate index

Because sloth's last index is 4, not 5, the JVM prints an error message. More specifically, it prints this:



```
Exception in thread "main"
java.lang.StringIndexOutOfBoundsException:
  String index out of range: 5
  at java.lang.String.charAt(String.java:558)
  at Test.main(Test.java:6)
```

The 5 refers to the specified index; it is "out of range."

The 6 refers to the line number in the program where the error occurred.

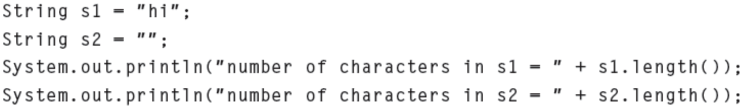
Ask: What is the computer trying to tell me?

At first, such error messages are intimidating and depressing, but eventually you'll learn to love them. Well, maybe not quite love them, but you'll learn to appreciate the -information they provide. They provide information about the type of error and where the error occurred. Try to view each error message as a learning opportunity! At this point, don't worry about understanding all the details in the above error message. Just focus on the two callouts and the lines that they refer to.

The above error is an example of a *runtime error*. A runtime error is an error that occurs while a program is running, and it causes the program to terminate abnormally. Said another way, it causes the program to *crash*.

The length Method

The length method returns the number of characters in a particular string. What does this code fragment print?



```
String s1 = "hi";
String s2 = "";
System.out.println("number of characters in s1 = " + s1.length());
System.out.println("number of characters in s2 = " + s2.length());
```

Because s1's string contains two

characters (‘h’ and ‘i’), the first print statement prints this:

```
number of characters in s1 - 2
```

In the code fragment, `s2` is initialized with the "" value. The "" value is commonly known as the *empty string*. An empty string is a string that contains no characters. Its length is zero. The second print statement prints this:

```
number of characters in s2 - 0
```

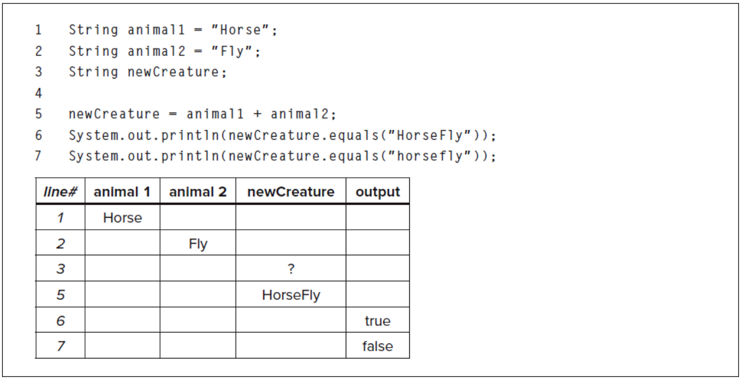
In calling the charAt method, you need to insert an argument (an index value) in the method call’s parentheses. For example, `animal.charAt(2).` On the other hand, in calling the length method, there’s no need to insert an argument in the method call’s parentheses. For example, `s1.length().` You may be thinking “With no argument, why bother with the parentheses?” In calling a method, you always need parentheses, even if they're empty. Without the parentheses, the compiler won't know that the method call is a method call.

The equals Method

To compare two strings for equality, it's necessary to step through the characters in both strings and compare same-positioned characters, one at a time. Fortunately, you don't have to write code to do that rather tedious comparison operation every time you want to see if two strings are equal. You just have to call the `equals` method, and it does the tedious comparison operation automatically, behind the scenes. More succinctly, the `equals` method returns `true` if two strings contain the exact same sequence of characters. It returns `false` otherwise.

Put yourself in the computer's place.

We recommend that you now go through a trace to make sure you thoroughly understand the `equals` method. See the code fragment in Figure 3.12. Try to trace the code fragment on your own prior to looking at the solution.



```
1   String animal1 = "Horse";
2   String animal2 = "Fly";
3   String newCreature;
4
5   newCreature = animal1 + animal2;
6   System.out.println(newCreature.equals("HorseFly"));
7   System.out.println(newCreature.equals("horsefly"));
```

line#	animal 1	animal 2	newCreature	output
1	Horse			
2		Fly		
3			?	
5			HorseFly	
6				true
7				false

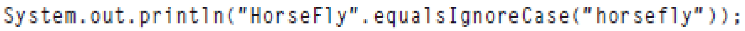
Figure 3.12 Code fragment that illustrates the `equals` method and its associated trace

Because newCreature contains the value “HorseFly”, the `equals` method returns a value of `true` when `newCreature` is compared to “HorseFly”. On the other hand, when `newCreature` is compared to -lowercase “horsefly”, the `equals` method returns a value of `false`.

The equalsIgnoreCase Method

Sometimes you might want to disregard uppercase versus lowercase when comparing strings. In other words, you might want “HorseFly” and “horsefly” to be considered equal. To test for case-insensitive equality, call the `equalsIgnoreCase` method.

What does this code fragment print?



```
System.out.println("HorseFly".equalsIgnoreCase("horsefly"));
```

Because `equalsIgnoreCase` considers “HorseFly” and “horsefly” to be equal, the code fragment prints `true`.

3.23 Input—the Scanner Class

Programs are normally a two-way street. They produce output by displaying something on the computer screen, and they read input from the user. Up to this point, all our Java programs and code fragments have gone just one way—they've displayed something on the screen, but they haven't read any input. With no input, our programs have been rather limited. In this section, we'll discuss how to get input from a user. With input, we'll be able to write programs that are much more flexible and useful.

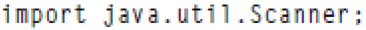
Ask: What If?

Suppose you're asked to write a program that calculates earnings for a retirement fund. If there's no input, your program must make assumptions about contribution amounts, years before retirement, and so on. Your program then calculates earnings based on those assumptions. Bottom line: Your no-input program calculates earnings for one specific retirement-fund plan. If input is used, your program asks the user to supply contribution amounts, years before retirement, and so forth. Your program then calculates earnings based on those user inputs. So which version of the program is better—the no-input version or the input version? The input version is better because it allows the user to plug in what-if scenarios. What happens if I contribute more money? What happens if I postpone retirement until I’m 90?

Input Basics

The Java API library provides a prebuilt class named Scanner, which allows you to get input from either a keyboard or a file. We describe file input in Chapter 16. Prior to that, when we talk about input, you should assume that we’re talking about keyboard input.

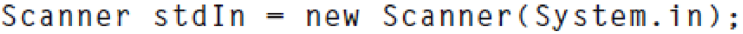
The Scanner class is not part of the core Java language. So if you use the Scanner class, you need to tell the compiler where to find it. You do that by importing the Scanner class into your program. More specifically, you need to include this import statement at the top of your program (right after your prologue section):



```
import java.util.Scanner;
```

We describe import details (like what is java.util?) in Chapter 5. For now, suffice it to say that you need to import the Scanner class in order to prepare your program for input.

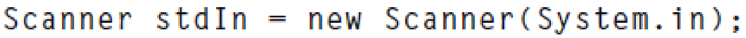
There's one more thing you need to do to prepare your program for input. Insert this statement at the top of your main method:



```
Scanner stdIn = new Scanner(System.in);
```

The new Scanner(System.in) expression creates an object. As you now know, an object stores a collection of data. In this case, the object stores characters entered by a user at a keyboard. The stdIn variable is a reference variable, and it gets initialized to the address of the newly created Scanner object. After the initialization, the stdIn variable allows you to perform input operations.

With the above overhead in place, you can read and store a line of input by calling the nextLine method like this:

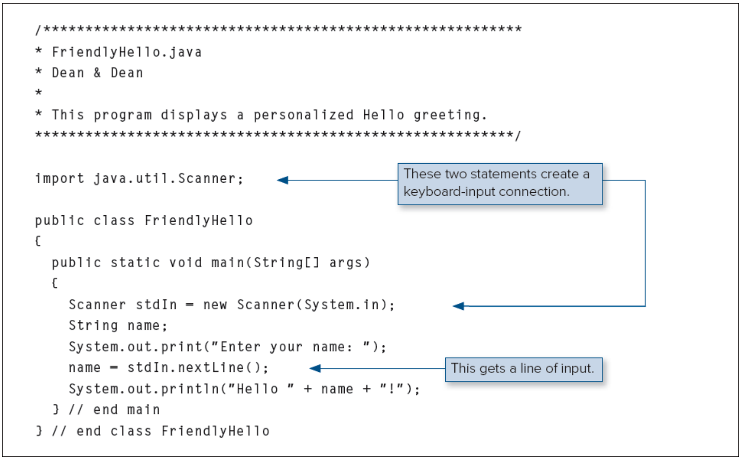


```
Scanner stdIn = new Scanner(System.in);
```

Let’s put what you’ve learned into practice by using the Scanner class and the nextLine method call in a complete program. See the FriendlyHello program in Figure 3.13. The program prompts the user to enter his or her name, saves the user’s name in a name variable, and then prints a greeting with the user’s name embedded in the greeting.

In the FriendlyHello program, note the “Enter your name: “ print statement. It uses a System.out.print statement rather than a System.out.println statement. Remember what the “ln” in println stands for? It stands for “line.” The System.out.println statement prints a message and then moves the screen’s cursor to the next line. On the other hand, the System.out.print statement prints a message and that’s it. The cursor ends up on the same line as the printed message (just to the right of the last printed character).

So why did we bother to use a print statement instead of a println statement for the “Enter your name:” prompt? Because users are used to entering input just to the right of a prompt message. If we used println, then the user would have to enter input on the next line. One additional item: We inserted a colon and a blank space at the end of the prompt. Once again, the rationale is that that’s what users are used to.



```
/*************************************************************
* FriendlyHello.java
* Dean & Dean
*
* This program displays a personalized Hello greeting.
*************************************************************/

import java.util.Scanner;

public class FriendlyHello
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String name;
    System.out.print("Enter your name: ");
    name = stdIn.nextLine();
    System.out.println("Hello " + name + "!");
  } // end main
} // end class FriendlyHello
```

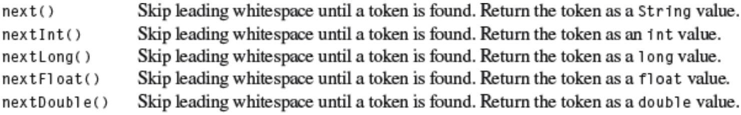
These two statements create a keyboard-input connection.

This gets a line of input.

Figure 3.13 FriendlyHello program

Input Methods

In the FriendlyHello program, we called the Scanner class's nextLine method to get a line of input. The Scanner class contains quite a few other methods that get different forms of input. Here are some of those methods:



`next()`	Skip leading whitespace until a token is found. Return the token as a `String` value.
`nextInt()`	Skip leading whitespace until a token is found. Return the token as an `int` value.
`nextLong()`	Skip leading whitespace until a token is found. Return the token as a `long` value.
`nextFloat()`	Skip leading whitespace until a token is found. Return the token as a `float` value.
`nextDouble()`	Skip leading whitespace until a token is found. Return the token as a `double` value.

The above descriptions need some clarification:

1. What is leading whitespace?

Whitespace refers to all characters that appear as blanks on a display screen or printer. This includes the space character, the tab character, and the newline character. The newline character is generated with the enter key. *Leading whitespace* refers to whitespace characters that are at the left side of the input.

2. The next method looks for a token. What is a token?

Think of a *token* as a word since the next method is usually used for reading in a single word. But more formally, a *token* is a sequence of non-whitespace characters. For example, “gecko” and “53B@a!” are tokens. But “Gila monster” forms two tokens, not one, because the space between “Gila” and “monster” signals the end of a token.

3. What happens if the user

provides invalid input for nextInt(), nextLong(), nextFloat(), or nextDouble()?

The JVM prints an error message and stops the program. For example, if a user enters hedgehog, 45g, or 45.0 in response to a nextInt() call, the JVM prints an error message and stops the program.

4. What is a Boolean value?

You’ll learn about Boolean values at the start of the next chapter, in Section 4.2.

Examples

To make sure you understand Scanner methods, study the programs in Figures 3.14 and 3.15. They illustrate how to use the nextDouble, nextInt, and next methods. Pay particular attention to the sample sessions. The sample sessions show what happens when the programs run with typical sets of input. In Figure 3.14, note the italics for 34.14 and 2. In Figure 3.15, note the italics for Ada Lovelace.[15] We italicize input values in order to distinguish them from the rest of the program. Be aware that the italicization is a pedagogical technique that we use for clarification purposes in the book. Input values are not really italicized when they appear on a computer screen.

```
/**********************************************************
* PrintPO.java
* Dean & Dean
*
* This program calculates and prints a purchase order amount.
**********************************************************/

import java.util.Scanner;

public class PrintPO
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double price;  // price of purchase item
    int qty;       // number of items purchased

    System.out.print("Price of purchase item: ");
    price = stdIn.nextDouble();
    System.out.print("Quantity: ");
    qty = stdIn.nextInt();
    System.out.println("Total purchase order = $" + price * qty);
  } // end main
} // end class PrintPO
```

Sample session:

```
Price of purchase item: 34.14
Quantity: 2
Total purchase order = $68.28
```

Figure 3.14 PrintPO program that illustrates nextDouble() and nextInt()

```
/*********************************************************
* PrintInitials.java
* Dean & Dean
*
* This program prints the initials for a user-entered name.
*********************************************************/

import java.util.Scanner;

public class PrintInitials
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String first;  // first name
    String last;   // last name

    System.out.print(
      "Enter your first and last name separated by a space: ");
    first = stdIn.next();
    last = stdIn.next();
    System.out.println("Your initials are " +
      first.charAt(0) + last.charAt(0) + ".");
  } // end main
} // end class PrintInitials
```

Sample session:

```
Enter first and last name separated by a space: Ada Lovelace
Your initials are AL.
```

Figure 3.15 PrintInitials program that illustrates next()

A Problem with the nextLine Method

⚠ The nextLine method and the other Scanner methods don’t play well together. It’s OK to use a series of nextLine method calls in a program. It’s also OK to use a series of nextInt, nextLong, nextFloat, nextDouble, and next method calls in a program. But if you use the nextLine method and the other Scanner methods in the same program, be careful. Here’s why.

The nextLine method is the only method that processes leading whitespace. The other methods skip it. Suppose you have a nextInt method call and the user types 25 and then presses the enter key. The nextInt method call reads the 25 and returns it. The nextInt method call does not read in the enter key’s newline character. Suppose that after the nextInt method call, you have a nextLine method call. The nextLine method call does not skip leading whitespace, so it’s stuck with reading whatever is left over from the previous input call. In our example, the previous input call left the enter key’s newline character. Thus, the nextLine call is stuck with reading it. Uh oh.

What happens if the nextLine method reads a newline character? It quits because it’s done reading a line (the newline character marks the end of a line, albeit a very short line). So the nextLine method call doesn’t get around to reading the next line, which is probably the line that the programmer intended it to read.

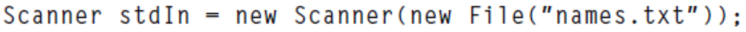
One solution to this nextLine problem is to include an extra nextLine method call whose sole purpose is to read in the leftover newline character.

Another solution is to use one Scanner reference variable for nextLine input (e.g., stdIn1) and another Scanner reference variable for other input (e.g., stdIn2). But for the most part, we'll try to steer clear of the problem altogether. We'll try to avoid nextLine method calls that follow one of the other Scanner method calls.

3.24 Simple File Input for Repetitive Testing During Program Development

As you progress through the book, you'll see that input from the computer keyboard and output to the computer screen is all the I/O you need to solve a vast array of complex problems. But if you have a large amount of input, it might be easier and safer to use a simple text processor to write that input once into a file and then reread it from that file each time you rerun the program.

Figure 3.16 shows an updated version of Figure 3.15’s PrintInitials program. This new version shows the original keyboard-input code commented out and replaced with code that reads the initials from the first name in a names.txt file. Note this line from the PrintInitials2 program:



```
Scanner stdIn = new Scanner(new File(”names.txt”));
```

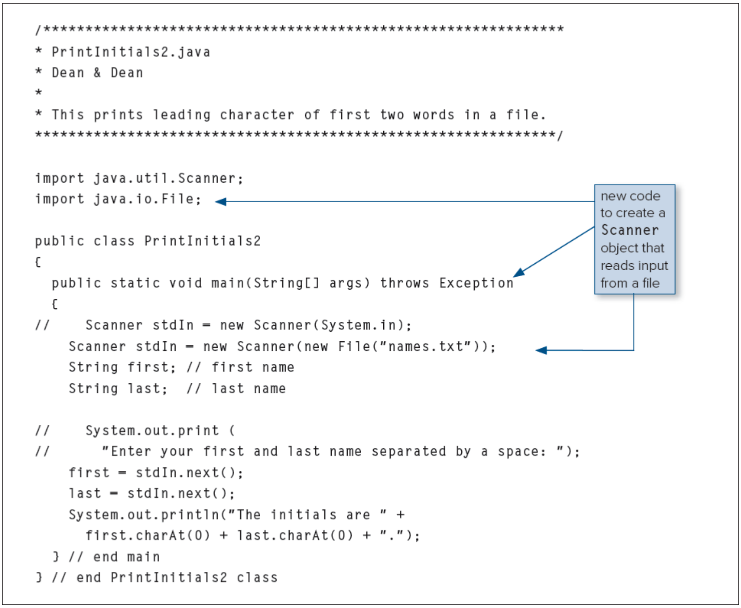
The new File("names.txt") code causes input to come from a file named names.txt. For comparison purposes, look at the commented-out Scanner object code, where System.in is used for keyboard input.

In the PrintInitials2 program’s new File("names.txt") code, note the word File. The File class is not part of the core Java language, so if you use it, you need to import it at the top of your program. More specifically, you need to do this:

```
import java.io.File;
```

The attempt to create a Scanner object with new Scanner(new File("names.txt")) might not work. For example, it won’t work if the operating system is unable to create the specified file. The Java compiler is a pessimist by nature, so, by default, it will generate a compilation error if you use the command new Scanner(new File("names.txt")) without some kind of “protective gear.” The simplest way to avoid the compilation error is to append throws Exception to the main method heading, like this:

```
public static void main(String[] args) throws Exception
```



```
/**************************************************************
* PrintInitials2.java
* Dean & Dean
*
* This prints leading character of first two words in a file.
**************************************************************/

import java.util.Scanner;
import java.io.File;

public class PrintInitials2
{
  public static void main(String[] args) throws Exception
  {
//    Scanner stdIn = new Scanner(System.in);
    Scanner stdIn = new Scanner(new File("names.txt"));
    String first; // first name
    String last;  // last name

//    System.out.print (
//      "Enter your first and last name separated by a space: ");
    first = stdIn.next();
    last = stdIn.next();
    System.out.println("The initials are " +
      first.charAt(0) + last.charAt(0) + ".");
  } // end main
} // end PrintInitials2 class
```

new code to create a Scanner object that reads input from a file

Figure 3.16 Updated version of the PrintInitials program that reads input from a file

The throws Exception clause tells the Java compiler that you’re aware that the JVM might not be able to do what you want. More specifically, you’re telling the compiler that an error (exception) might be generated (thrown). The throws Exception clause won’t magically guarantee that the file will be created successfully, but if you don’t append it to the main method heading, your program won’t even be able to try. Why? Because it won’t compile. Appending throws Exception to the main method header is a “simple and dirty” way to force successful compilation.

Chapter 15 describes a more responsible treatment of exceptions. If you want to use this more responsible treatment when you write to or read from a file, look at the HTMLGenerator program in Section 16.2. It contains code that you can use as a template. That code is more complicated, however, than just appending throws Exception to main.

At this point, some readers might want to apply what they’ve learned to an object-oriented programming (OOP) context. OOP is the idea that programs should be organized into objects. You’re not required to learn about OOP just yet, but if you can’t wait, you can find such details in Chapter 6, Sections

6.1 through 6.8.

3.25 GUI Track: Input and Output with Dialog Boxes (Optional)

This section presents three prewritten JavaFX classes (Alert, TextInputDialog, and ChoiceDialog) whose common features descend from a shared ancestral class, Dialog. A *dialog box* (or *dialog* for short) is a window that is rather limited in terms of what it does and how the programmer can customize it. These three dialog types perform different tasks. An Alert displays output. A TextInputDialog displays a question and provides an input field. A ChoiceDialog provides a pull-down menu with optional choices. After a program begins execution, these GUI dialogs lift the user interface out of the pedestrian realm of the console and enable user inputs and outputs to be in the form of a graphical user interface.

Starting a JavaFX GUI Application

This section's DialogDemo program illustrates all three dialog types along with corresponding output displays. The import statements in Figure 3.17a provide access to the prewritten JavaFX application code our program leverages. Like other JavaFX programs, this program extends JavaFX's Application class. Any extension of the Application class (like this one) automatically creates a Stage object (a GUI window frame) and then calls its start method with that object as an argument. Every JavaFX program must include an explicit programmer-defined start method.

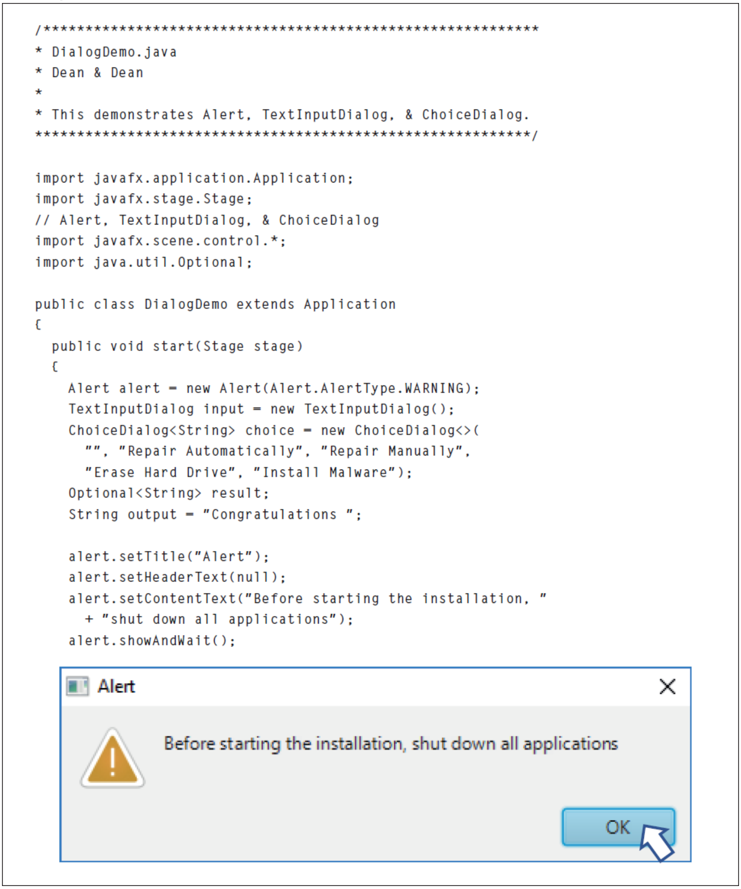
The start method's first statement

creates an instance of the prewritten Alert class and specifies its type as INFORMATION. The possible values for Alert.AlertType are:

CONFIRMATION ?, ERROR ✕, INFORMATION ℹ, WARNING ⚠, and NONE.

The Alert.AlertType.NONE value requires the programmer to specify one or more buttons, which you'll learn to do later. The other AlertType values do not require the programmer to specify a button. They automatically supply an OK button by default.

The next two statements in Figure 3.17a create a TextInputDialog (for user input from a text box) and a ChoiceDialog (for user input from a list of choices). The next statement declares a result variable with the special type Optional<String>. That declaration enables the result variable to receive user input from a GUI window. The statement after that creates and initializes a String output variable to accumulate user inputs for eventual output.



```
/*************************************************************
* DialogDemo.java
* Dean & Dean
*
* This demonstrates Alert, TextInputDialog, & ChoiceDialog.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
// Alert, TextInputDialog, & ChoiceDialog
import javafx.scene.control.*;
import java.util.Optional;

public class DialogDemo extends Application
{
  public void start(Stage stage)
  {
    Alert alert = new Alert(Alert.AlertType.WARNING);
    TextInputDialog input = new TextInputDialog();
    ChoiceDialog<String> choice = new ChoiceDialog<>(
      "", "Repair Automatically", "Repair Manually",
      "Erase Hard Drive", "Install Malware");
    Optional<String> result;
    String output = "Congratulations ";

    alert.setTitle("Alert");
    alert.setHeaderText(null);
    alert.setContentText("Before starting the installation, "
      + "shut down all applications");
    alert.showAndWait();
```

Before starting the installation, shut down all applications

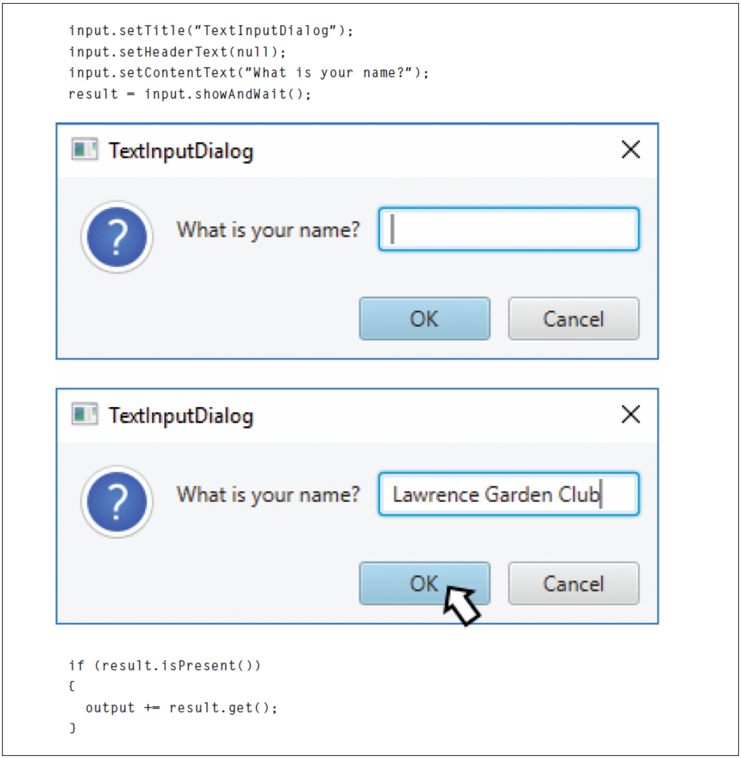
Figure 3.17a First part of DialogDemo program and its generated display

The setTitle method sets the title at the left side of the window's *title bar* (the banner at the top of the window). The setHeaderText method's null argument suppresses a heading that would otherwise appear in the top half of the window. The setContentText method sets text to be displayed in the center of the remaining window. As an alternative to calling setContentText, you can supply the content text string as a second argument in the Alert constructor call.

The showAndWait method displays the Alert dialog and pauses (blocks) all other program operations until the user closes the window by clicking either the OK button or the dialog box's X closeout button. When the dialog closes, execution resumes.

Input and Choice Dialogs

Figure 3.17b shows additional code from the DialogDemo program and the dialog it generates. Its input variable holds a reference to a TextInputDialog object. With our previous code that generated the Alert box, there was no returned value. But with the TextInputDialog, the user enters a value into the dialog’s text box, the showAndWait method call returns that value, and it gets assigned to a result variable. After an if statement checks whether a user response has occurred, additional code adds result's content to an output string. (Because the result variable is not itself a String, we need for result to call its get method to retrieve its String content.)



```
input.setTitle(”TextInputDialog”);
input.setHeaderText(null);
input.setContentText(”What is your name?”);
result = input.showAndWait();
```

TextInputDialog
What is your name?
OK
Cancel
TextInputDialog
What is your name?

Lawrence Garden Club

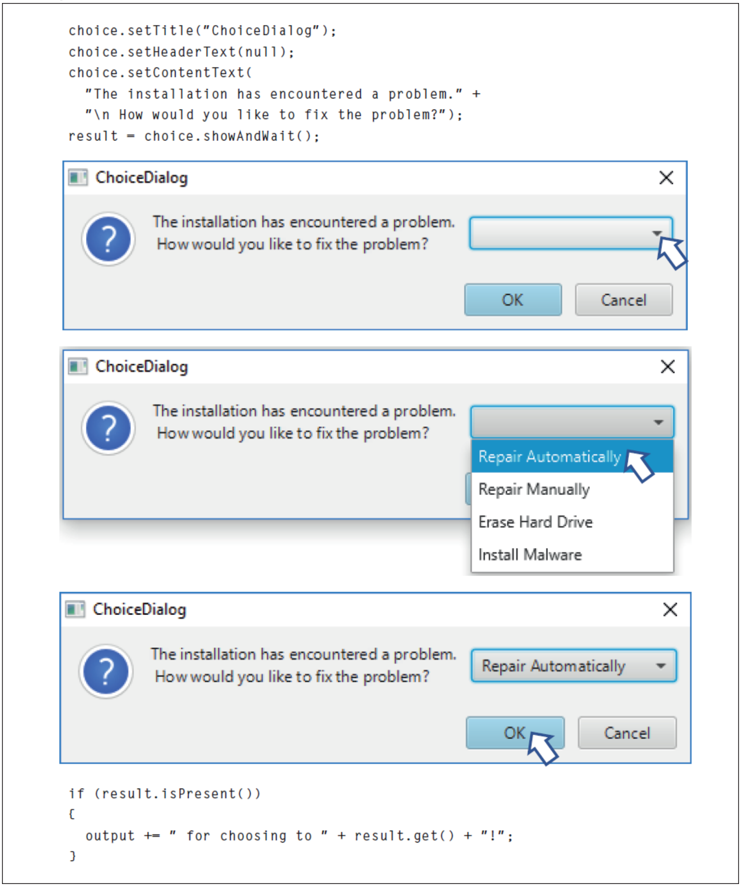
OK
Cancel

```
if (result.isPresent())
{
  output += result.get();
}
```

Figure 3.17b Second part of DialogDemo program with its generated display

Figure 3.17c shows still more code from the DialogDemo program and the dialog it generates. Its choice variable holds a reference to a ChoiceDialog object. As before, the showAndWait method displays the dialog and pauses until the user closes it by clicking OK. When the dialog closes and returns a result to the result variable, result's contents are appended to the output string.

Page 110



```
choice.setTitle("ChoiceDialog");
choice.setHeaderText(null);
choice.setContentText(
  "The installation has encountered a problem." +
  "\n How would you like to fix the problem?");
result = choice.showAndWait();
```

ChoiceDialog

The installation has encountered a problem.
How would you like to fix the problem?

ChoiceDialog

The installation has encountered a problem.
How would you like to fix the problem?

Repair Automatically
Repair Manually
Erase Hard Drive
Install Malware

ChoiceDialog

The installation has encountered a problem.
How would you like to fix the problem?

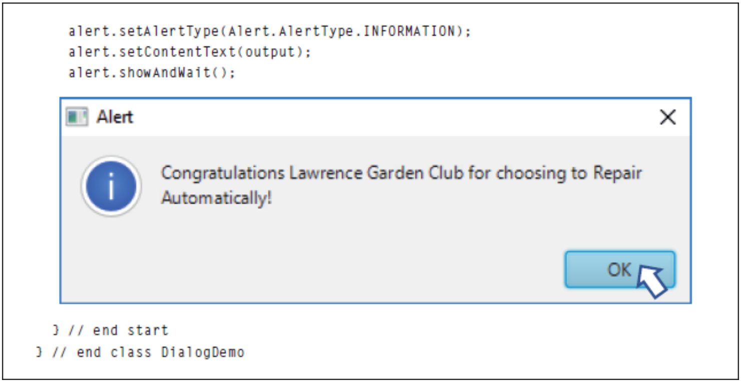
Repair Automatically

```
if (result.isPresent())
{
  output += " for choosing to " + result.get() + "!";
}
```

Figure 3.17c Third part of DialogDemo program with its generated displays

Figure 3.17d shows the final code from the DialogDemo program and the dialog it generates. It reuses the alert variable from earlier. This time, it calls

Alert's setAlertType with an INFORMATION argument, so its dialog displays with an information icon.



```
    alert.setAlertType(Alert.AlertType.INFORMATION);
    alert.setContentText(output);
    alert.showAndWait();
```

Congratulations Lawrence Garden Club for choosing to Repair Automatically!

```
  } // end start
} // end class DialogDemo
```

Figure 3.17d Final part of DialogDemo program with its generated displays

Summary

- Comments are used for improving a program’s readability/understandability.
- The System.out.println method prints a message and then moves the screen’s cursor to the next line. The System.out.print method prints a message and leaves the

cursor on the same line as the printed message.

- Variables can hold only one type of data item, and that type is defined with a variable declaration statement.

- An assignment statement uses the = operator, and it puts a value into a variable.

- An initialization statement is a combination of a declaration statement and an assignment statement. It declares a variable's type and also gives the variable an initial value.

- Variables that hold whole numbers should normally be declared with the int data type or the long data type.

- Variables that hold floating-point numbers should normally be declared with the double data type. If you're sure that a variable is limited to small floating-point numbers, it's OK to use the float data type.

- Named constants use the final modifier.

- There are two types of integer division. One type finds the quotient (using the / operator). The other type finds the remainder (using the % operator).

- Expressions are evaluated using a set of well-defined operator precedence rules.

- The cast operator allows you to return a different-data-type version of a given value.

- Use an escape sequence (with a backslash) to print hard-to-print characters such as the tab character.

- A reference variable stores a memory address that points to an object. An object is a collection of related data wrapped in a protective shell.
- The String class provides methods that can be used for string processing.
- The Scanner class provides methods that can be used for input.

Review Questions

§3.2 "I Have a Dream" Program

1. What does this chapter's Dream.java program do?

2. What are the filename extensions for Java source code and bytecode, respectively?

§3.3 Comments and Readability

3. Why does source code have comments?

§3.4 The Class Heading

4. For a file with a public class, the program's filename must match the program's class name except that the filename has a .java extension added to it. (T / F)

5. Standard coding conventions dictate that class names start with a lowercase first letter. (T / F)

6. In Java, the case of a character does matter. (T / F)

§3.5 The main Method's Heading

7. A program's start-up method, main, should be in a class that is public. (T / F)

8. The main method itself must be public. (T / F)

9. From your memory alone (don't look for the answer in the book), write the main method heading.

§3.6 Braces

10. Identify two types of groupings that must be enclosed in braces.

§3.7 System.out.println

11. From your memory alone (don’t look for the answer in the book), write the statement that tells the computer to display this string of text:

Here is an example

§3.9 Identifiers

12. List all of the types of characters that may be used to form an identifier.

13. List all of the types of characters that may be used as the first character of an identifier.

§3.10 Variables

14. You should abbreviate variable names by omitting vowels, in order to save space. (T / F)

15. Why is it good practice to use a separate line to declare each separate variable?

§3.11 Assignment Statements

16. There must be a semicolon after every assignment statement. (T / F)

§3.12 Initialization Statements

17. Initialization “kills two birds with one stone.” What are the “two birds”?

§3.13 Numeric Data Types—int, long, float, double

18. The most appropriate type to use for financial accounting is

____________ .

19. For each statement, specify true or false:

- **a)** 1234.5 is a floating-point number. (T / F)
- **b)** 1234 is a floating-point number. (T / F)
- **c)** 1234. is a floating-point number. (T / F)

20. If you try to assign an int value into a double variable, the computer automatically makes the conversion without complaining, but if you try to assign a double value into an int variable, the compiler generates an error. Why?

§3.14 Constants

21. For each statement, specify true or false:

- **a)** 0.1234 is a float. (T / F)

b) 0.1234f is a float. (T / F)

c) 0.1234 is a double. (T / F)

d) 1234.0 is a double. (T / F)

22. What modifier specifies that a variable’s value is fixed/constant?

§3.15 Arithmetic Operators

23. What is the remainder operator?

24. Write the following mathematical expressions as legal Java expressions:

a) $\frac{3x-1}{x^2}$

b) $\frac{1}{2}+\frac{1}{xy}$

§3.16 Expression Evaluation and Operator Precedence

25. Assume this:

```
int m = 3, n = 2;
double x = 7.5;
```

Evaluate the following expressions:

a) (7 - n) % 2 * 7.5 + 9

b) (4 + n / m) / 6.0 * x

§3.17 More Operators: Increment, Decrement, and Compound Assignment

26. Write the shortest Java statement that increments count by 1.

27. Write the shortest Java statement that decrements count by 3.

28. Write the shortest Java statement that multiplies number by (number - 1) and leaves the product in number.

§3.18 Tracing

29. What does it mean if a variable contains garbage?

30. In a trace listing, what are line numbers for?

§3.19 Type Casting

31. Write a Java statement that assigns the double variable, myDouble, to the int variable, myInteger.

§3.20 char Type and Escape Sequences

32. What's wrong with the following initialization?

```
char letter = "y";
```

33. If we try to put a quotation mark (") somewhere inside a string literal to be printed, the computer interprets the quotation mark as the end of the string literal. How can we overcome this problem and force the computer to recognize the quotation mark as something we want to print?

34. When describing the location of a file or directory, computers use directory paths. In Windows environments, use the backslash character (\) to separate directories and files within a directory path. If you need to print a directory path within a Java program, how should you write the backslash character?

§3.21 Primitive Variables Versus Reference Variables

35. The type name for a primitive type is not capitalized, but the type name for a reference type is usually capitalized. (T / F)

36. List the primitive types this chapter describes, in the following categories:

- **a)** Integer numbers
- **b)** Floating-point numbers
- **c)** Individual text characters and special symbols

§3.22 Strings

37. What two operators perform

string concatenation, and what's the difference between these operators?

38. What method can be used to retrieve a character at a specified position within a string?

39. What two methods can be used to compare strings for equality?

§3.23 Input—the Scanner class

40. What is whitespace?

41. Write the statement that you must put before any other code to tell the compiler that you will be using the Scanner class.

42. Write the statement that creates a connection between your program and the computer's keyboard.

43. Write a statement that inputs a line of text from the keyboard and puts it into a variable named line.

44. Write a statement that inputs a double number from the keyboard and puts it into a variable named number.

Exercises

1. [after §3.3] Illustrate the two ways to provide comments in a Java program by writing the following as comments. Use a different comment format for each of the two sentences.

This is a relatively long sentence that, if embedded in a Java program, would confuse the Java compiler and cause an error to be generated unless, of course, it was commented out.

This is a relatively short sentence.

2. [after §3.5] Why does public static void main(string[] args) generate an error?

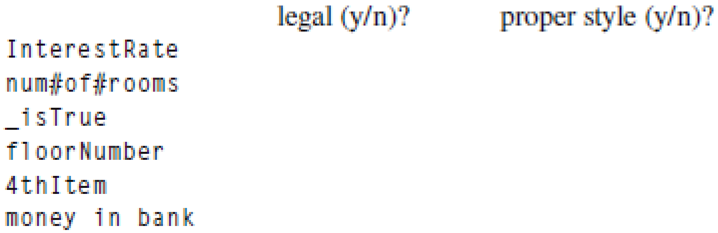
3. [after §3.6] Identify two uses for braces.

4. [after §3.8] What does each of the following programs do?

- **a)** The Java compiler?
- **b)** Java virtual machine (JVM)?

5. [after §3.9] In Java, how do we write multiple-word identifiers?

6. [after §3.10] For each of the below variable names, indicate (with y or n) whether it's legal and whether it uses proper style. Note: You may skip the style question for illegal variable names because style is irrelevant in that case.



	legal (y/n)?	proper style (y/n)?
`InterestRate`		
`num#of#rooms`		
`_isTrue`		
`floorNumber`		
`4thItem`		
`money in bank`		

7. [after §3.10] Each variable must be declared in a separate statement. (T / F)

8. [after §3.13] If an algebraic expression includes a number like 0.47, what type does the computer assume it is, a float or a double?

9. [after §3.14] Use 1.7320508075688772 in the declaration of a named constant that represents the square root of 3.

10. [after §3.15] Write the following mathematical expressions as legal Java expressions:

a) $\left(\frac{4-n}{5}\right)^3$

b) $\frac{7.5x-(3.5+y)}{18x}$

11. [after §3.16] Assume this:

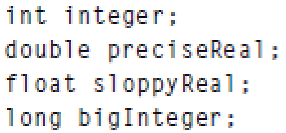
```
int a = 7;
double b = 0.4;
```

Evaluate each of the following expressions by hand. Show your work, using a separate line for each evaluation step. Check your work by writing and executing a program that evaluates these expressions and outputs the results.

- **a)** a + 3 / a
- **b)** 25 / ((a - 4) * b)
- **c)** a / b * a
- **d)** a % 2 - 2 % a

12. [after §3.19] Type Casting:

Assume the following declarations:



```
int integer;
double preciseReal;
float sloppyReal;
long bigInteger;
```

Rewrite those of the following statements which would generate a compile-time error using an appropriate cast that makes the error go away. Do not provide a cast for any statement which the compiler automatically promotes.

```
integer = preciseReal;
sloppyReal = integer;
bigInteger = sloppyReal;
preciseReal = bigInteger;
integer = bigInteger;
sloppyReal = bigInteger;
bigInteger = preciseReal;
bigInteger = integer;
integer = sloppyReal;
preciseReal = sloppyReal;
sloppyReal = preciseReal;
preciseReal = integer;
```

13. [after §3.20] Assuming that tab stops are 4 columns apart, what output does the following statement generate?

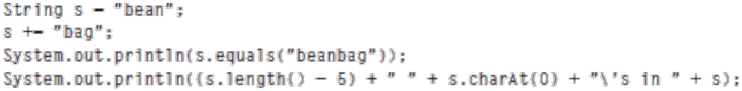
```
System.out.println("\"files:\"\n\tE:\\code\\Hello.java");
```

14. [after §3.21] What’s wrong with this statement?

```
string programmingPioneer = ”Admiral Grace Hopper”;
```

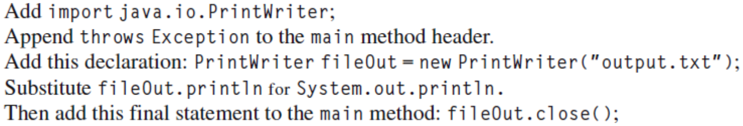
15. [after §3.22] Assume that you have a string variable named lastName. Provide a code fragment that prints lastName's first character.

16. [after §3.22] What does this code fragment print?



```
String s = "bean";
s += "bag";
System.out.println(s.equals("beanbag"));
System.out.println((s.length() - 5) + " " + s.charAt(0) + "\'s in " + s);
```

17. [after §3.23] Modify the PrintInitials program so that it takes input from the keyboard but prints output to a file named output.txt. Use these suggestions:

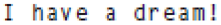


Add `import java.io.PrintWriter;`
Append `throws Exception` to the `main` method header.
Add this declaration: `PrintWriter fileOut = new PrintWriter("output.txt");`
Substitute `fileOut.println` for `System.out.println`.
Then add this final statement to the `main` method: `fileOut.close();`

Review Question Solutions

---

1. It generates the output:



```
I have a dream!
```

2. The Java source code extension is .java. The bytecode extension is .class.

3. Source code has comments to help Java programmers recall or determine how a program works. (Comments are ignored by the computer, and they are not accessible to ordinary users.) The initial comment block includes the file name as a continuous reminder to the programmer. It contains program authors, for help and reference. It may include date and version number to identify context. It includes a short description to facilitate rapid understanding. Punctuation comments like // end class *classname* help keep a reader oriented. Special comments identify variables and annotate obscure formulas.

4. True. If a file has a public class, the filename must equal this class name.

5. False. Class names should start with an uppercase first letter.

6. True. Java is case sensitive.

Changing the case of any letter creates a completely different identifier.

7. True.

8. True. Otherwise, the startup procedure cannot be accessed.

9. public static void main(String[] args)

10. One must use braces for (1) all the contents of a class and (2) all the contents of a method.

11. System.out.println("Here is an example");

12. Uppercase characters, lowercase characters, numbers, underscore, and dollar sign.

13. Uppercase characters, lowercase characters, underscore, and dollar sign. No numbers.

14. False: In source code, saving space is not as important as good communication. Weird abbreviations are hard to say and not as easy to remember as real words.

15. If each variable is on a separate line, each variable has space at the right for an elaborating comment.

16. True.

Page 117

17. Variable declaration and assigning a value into the variable.

18. Type double, or type long, with value in cents.

19. **a**) True; **b**) False; **c**) True

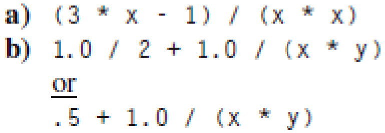
20. Assigning an integer value into a floating-point variable is like putting a small object into a large box. The int type goes up to approximately 2 billion. It’s easy to fit 2 billion into a double “box” because a double goes all the way up to 1.8×10^{308}. On the other hand, assigning a floating-point value into an integer variable is like putting a large object into a small box. By default, that's illegal.

21. **a)** False; **b)** True; **c)** True; **d)** True

22. The final modifier specifies that a variable's value is fixed/constant.

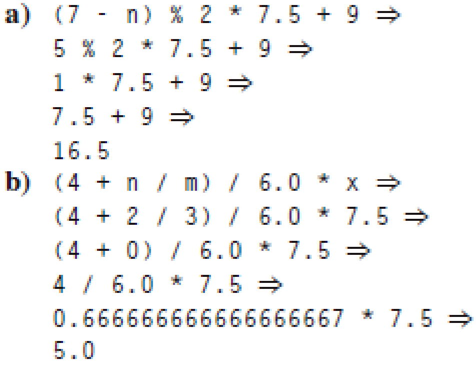
23. The remainder operator is a percent sign: %.

24. Write the following mathematical expressions as legal Java expressions:



```
a) (3 * x - 1) / (x * x)
b) 1.0 / 2 + 1.0 / (x * y)
   or
   .5 + 1.0 / (x * y)
```

25. Expression evaluation:



```
a) (7 - n) % 2 * 7.5 + 9 ⇒
   5 % 2 * 7.5 + 9 ⇒
   1 * 7.5 + 9 ⇒
   7.5 + 9 ⇒
   16.5
b) (4 + n / m) / 6.0 * x ⇒
   (4 + 2 / 3) / 6.0 * 7.5 ⇒
   (4 + 0) / 6.0 * 7.5 ⇒
   4 / 6.0 * 7.5 ⇒
   0.6666666666666667 * 7.5 ⇒
   5.0
```

26. count+ +;

27. count -= 3;

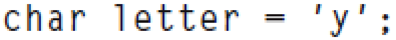
28. number *= (number - 1);

29. For variables declared without an initialization, the initial value is referred to as *garbage* because its actual value is unknown. Use a question mark to indicate a garbage value.

30. Line numbers tell you which statement in the code generates the current trace results.

31. myInteger = (int) myDouble;

32. The variable, letter, is of type char, but the double quotes in "y" specify that the initial value has type String, so the types are incompatible. It should be written:



```
char letter = 'y';
```

33. To print a double quotation mark, put a backslash in front of it, that is, use \".

34. To print a backslash, use two backslashes, that is, use \\.

35. True.

Page 118

36. List the primitive types this chapter describes, in the following categories:

- **a)** Integer numbers: int, long
- **b)** Floating-point numbers: float, double
- **c)** Individual text characters and special symbols: char

37. The + and += operators perform *concatenation*. The + operator does not update the operand at its left. The += operator does update the operand at its left.

38. The charAt method can be used to retrieve a character at a specified position within a string.

39. The equals and equalsIgnoreCase methods can be used to compare strings for equality.

40. Whitespace characters are the characters associated with the spacebar, the tab key, and the enter key.

41. import java.util.Scanner;

42. Scanner stdIn = new Scanner(System.in);

43. line = stdIn.nextLine();

44. number = stdIn.nextDouble();

---

[1] Dr. Martin Luther King, Jr., presented his famous “I have a dream” speech on the steps of the Lincoln Memorial as part of an August 28, 1963 civil rights march on Washington, D.C. The speech supported desegregation and helped spur passage of the 1964 Civil Rights Act.

[2] A *compiler*, defined in Chapter 1, is a special program that converts a source

code program into an executable program. An *executable program* is a program that the computer can execute directly.

[3] In Java, reserved words and keywords are the same. But in some programming languages, there is a subtle difference. In those languages, both terms refer to words that are defined by the programming language, but keywords can be redefined by the programmer, while reserved words cannot be redefined by the programmer.

[4] To access a non-static method (more formally called an *instance method*), you must first instantiate an object. We describe object instantiation in Chapter 6.

[5] Although the main method takes arguments, it's rare for the main method to use those arguments. Most of this book's programs do not use the

main method's arguments.

[6] *Bytecode*, defined in Chapter 1, is a binary-encoded version of the source code. The computer cannot execute source code, but it can execute bytecode.

[7] If you'd like a more detailed discussion of tracing, see Chapter 2, Section 2.11.

[8] If you run the code fragment on a computer, you'll see a .0 at the end of the output (Bonus = 1000.0). The .0 should make sense when you learn about mixed expressions and promotion later in this chapter.

[9] John Doe is commonly used as a filler in the United States and Great Britain when a person's real name is unknown. We use it here as a default value for a student's name. It serves as an indication that the student's real name has not yet been filled in.

[10] The suggestion to use an int for efficiency reasons is valid, but be aware that the speed difference is only occasionally noticeable. It’s noticeable only if you have lots of long numbers and you have a small amount of available memory, such as when you’re running a relatively large program on a cell phone.

[11] A float variable can store positive values between $1.2 \times 10-^{38}$ and $3.4 \times 10+^{38}$ and negative values between $-3.4 \times 10+^{38}$ and $-1.2 \times 10-^{38}$. A double variable can store positive values between $2.2 \times 10-^{308}$ and $1.8 \times 10+^{308}$ and negative values between $-1.8 \times 10+^{308}$ and $-2.2 \times 10-^{308}$.

[12] “Champing at the bit” is an idiom referring to the habit of horses chewing their bits when they’re (supposedly) impatient. A bit is a small metal rod placed in a horse’s mouth

which, when attached to the reins, enables the rider to guide the horse. To our equine readers: we apologize in advance for bringing up what might be an uncomfortable subject.

[13] The word “atom” comes from the Greek *a-tomos* and means indivisible. In 1897, J. J. Thomson discovered one of the atom’s components—the electron—and thus dispelled the notion of an atom’s indivisibility. Nonetheless, as a holdover from the original definition of atom, the term “atomic” still refers to something that is inherently indivisible.

[14] Explaining the Calendar class in depth is beyond the scope of this chapter. If you want an in-depth explanation, you could go to https://docs.oracle.com/en/java/javase/13/docs/api/java.base/java/util/package-summary.html and search for Calendar.

[15] Ada Lovelace, 1815–1852, is widely credited as the world’s first computer programmer. She wrote a program to calculate Bernoulli numbers as a means to show the usefulness of her friend Charles Babbage’s Analytical Engine, which is considered to be the world’s first design of a programmable computer. Although Babbage never completed the construction of his Analytical Engine, Lovelace recognized its enormous potential to solve problems. In her honor, the U.S. Department of Defense created Ada, a popular programming language that has been praised for its flexibility but criticized for its relative slowness.

Control Statements

Objectives

- Learn how to use if statements to alter a program’s sequence of execution.
- Become familiar with Java’s comparison and logical operators, and learn how to use them to describe complex conditions.

- Learn how to use switching constructs (the switch statement and the switch expression) to alter a program’s sequence of execution.
- Recognize repetitive operations, understand the various kinds of looping that Java supports, and learn how to select the most appropriate type of loop for each problem that requires repetitive evaluation.
- Be able to trace a looping operation.
- Learn how and when to nest a loop inside another loop.
- Learn how to use boolean variables to make code more elegant.
- Learn how to validate input data.
- Optionally, learn how to simplify complicated logical expressions.

Outline

4.1 Introduction 120

4.2 Conditions and Boolean Values 120

4.3 if Statements 121

4.4 Logical Operator 124

4.5 || Logical Operator 129

4.6 ! Logical Operator 131

4.7 Switching Constructs 132

4.8 while Loop 138

4.9 do Loop 142

4.10 for Loop 144

4.11 Solving the Problem of Which Loop to Use 149

4.12 Nested Loops 150

4.13 boolean Variables 152

4.14 Input Validation 156

4.15 Problem Solving with Boolean Logic (Optional) 157

4.1 Introduction

In Chapter 3, we kept things simple and wrote pure sequential programs. In a pure sequential program, statements execute in the sequence/order in which they are written; that is, after executing a statement, the computer executes the statement that immediately follows it. Pure sequential programming works well for trivial problems, but for anything substantial, you'll need the ability to execute in a nonsequential fashion. For example, if you're writing a recipe-retrieval program, you probably don't want to print all the program's recipes, one after another. You'll want to execute the chocolate chip cookie print statements if the user indicates an affinity for chocolate chip cookies, and you'll want to execute the crab quiche print statements if the user indicates an affinity for crab quiche. That sort of functionality requires the use of *control statements*. A control statement controls the order of execution of other statements. In Chapter 2, you used pseudocode if and while statements to control the order of execution within an algorithm. In this chapter, you'll use Java if and while statements, plus a few additional Java statements, to control the order of execution within a program.

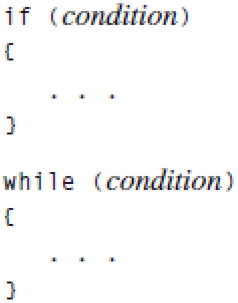
In controlling the order of execution, a control statement uses a condition (a question) to decide which way to go. We start Chapter 4 with an overview of Java conditions. We then describe Java's control statements—the if statement, the switch statement, a statement with an embedded switch expression, the while loop, the do loop, and the for loop. Along the way, we describe Java's logical operators &&, ||, and !, which are needed when dealing with more complicated conditions. We conclude the chapter with several loop-related concepts—nested loops, input validation, and boolean variables. Good stuff!

4.2 Conditions and Boolean Values

In Chapter 2's flowcharts, we used diamond shapes to represent logical decision points—points where control flow went either one way or another. Into those diamonds we inserted various abbreviated questions like "ceoSalary greater than $500,000?" and "count less than or equal to 100?" Then we labeled alternate paths away from those diamonds with "yes" or "no" answers to those questions. In

Chapter 2’s pseudocode, we used “if” and “while” clauses to describe logical conditions. Examples included “if shape is a circle,” “if grade is greater than or equal to 60,” and “while score is not equal to —1.” We considered pseudocode conditions to be either “true” or “false.”

Informal condition expressions like these are fine for flowcharts and pseudocode, but when you start writing real Java code, you must be precise. The computer interprets each “if” condition or loop condition as a two-way choice. What are the two possible values recognized by Java? They are the Java values true and false. These values, true and false, are called *Boolean* values, after George Boole, a famous 19th-century logician. Throughout the rest of this chapter, you’ll see if statements and loop statements where *conditions* appear as little fragments of code within a pair of parentheses, like this:



```
if (condition)
{
   . . .
}

while (condition)
{
   . . .
}
```

Whatever is in the places marked by *condition* always evaluates to either true or false.

Typically, each condition involves some type of comparison. With pseudocode, you may use words to describe comparisons, but in real Java code, you must use special comparison operators. Comparison operators (also called *equality* and relational operators) are like mathematical operators in that they link adjacent operands, but instead of combining the operands in some way, they compare them. When a mathematical operator combines two numbers, the combination is a number like the operands being combined. But when a comparison operator compares two numbers, the result is a different type. It is not a number like the operands being compared. It is a Boolean truth value—either true or false.

Here are Java's comparison operators:

```
==, !=, <, >, <=, >=
```

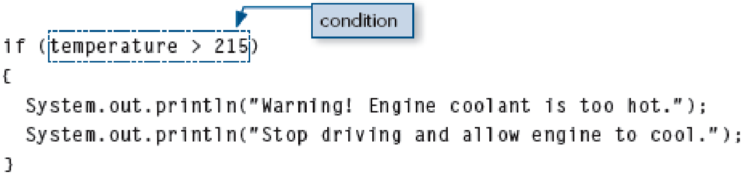
The == operator tests whether two values are equal. Notice that this symbol uses two equals signs! This is different from the single equals sign that we all use instinctively to represent equality. Why does Java use two equals signs for equality in a comparison? It's because Java already uses the single equals sign for assignment, and context is not enough to distinguish assignment from equality. Do not try to use a single = for comparison! The Java compiler will not like it. ⚠

The != operator tests whether two

values are unequal. As you'd expect, the < operator tests whether a value on the left is less than a value on the right. The > operator tests whether a value on the left is greater than a value on the right. The <= operator tests whether a value on the left is less than or equal to a value on the right. The >= operator tests whether a value on the left is greater than or equal to a value on the right. The result of any one of these tests is always either true or false.

4.3 if Statements

Now, let's look at a simple example of the condition in an if statement. Here's a simple if statement that checks a car's temperature gauge value and prints a warning if the temperature is above 215 degrees:



```
if (temperature > 215)
{
  System.out.println("Warning! Engine coolant is too hot.");
  System.out.println("Stop driving and allow engine to cool.");
}
```

The condition uses the > operator to generate a true value if the temperature is above 215 degrees or a false value if it is not. The subordinate statements (the statements in the braces) execute only if the condition generates a true value.

Syntax

In the above example, note the parentheses around the condition. Parentheses are required whenever you have a condition, regardless of whether it’s for an if statement, a while loop, or some other control structure. Note the braces around the two subordinate print statements. Use braces to surround statements that are logically inside something else. For example, braces are required below the main method's heading and at the bottom of the main method because the statements inside the braces are logically inside the main method. Likewise, you should use braces to surround the statements that are logically inside an if statement. To emphasize the point that statements inside braces are logically inside something else, you should always indent statements that are inside braces. Because this is so important, we'll say it again: Always indent when you're inside braces!

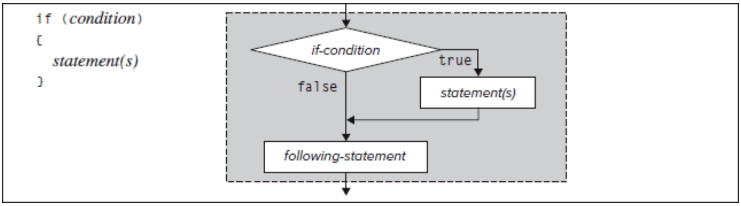
When an if statement includes two or more subordinate statements, you must enclose the subordinate statements in braces. Said another way, you must use a *block*. A block, also called a *compound statement,* is a set of zero or more statements surrounded by braces. A block can be used anywhere a standard statement can be used. If you don’t use braces for the if statement’s two subordinate statements, the computer considers only the first statement to be subordinate to the if statement. When there is supposed to be just one subordinate statement, you’re not required to enclose it in braces, but we recommend that you do so anyway. That way, you won’t get into trouble if you come back later and want to insert additional subordinate statements at that point in your program. ⚠

Three Forms of the if Statement

There are three basic forms for an if statement:

- “if”—use when you want to do one thing or nothing.
- “if, else”—use when you want to do one thing or another thing.
- “if, else if”—use when there are three or more possibilities.

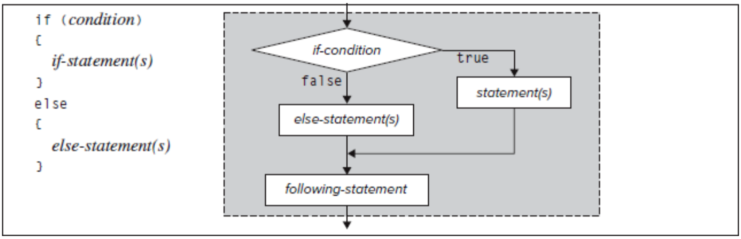
Chapter 2 presented pseudocode versions of these forms. Figures 4.1, 4.2, and 4.3 show Java forms.



```
if (condition)
{
  statement(s)
}
```

if-condition
true
false
statement(s)
following-statement

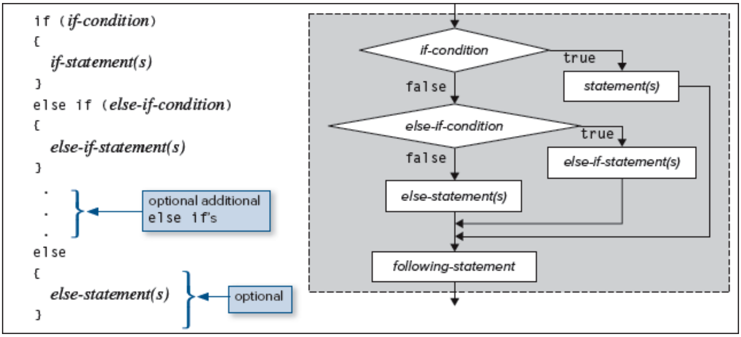
Figure 4.1 Syntax and semantics for the simple "if" form of the if statement



```
if (condition)
{
  if-statement(s)
}
else
{
  else-statement(s)
}
```

if-condition
true
false
statement(s)
else-statement(s)
following-statement

Figure 4.2 Syntax and semantics for the "if, else" form of the if statement



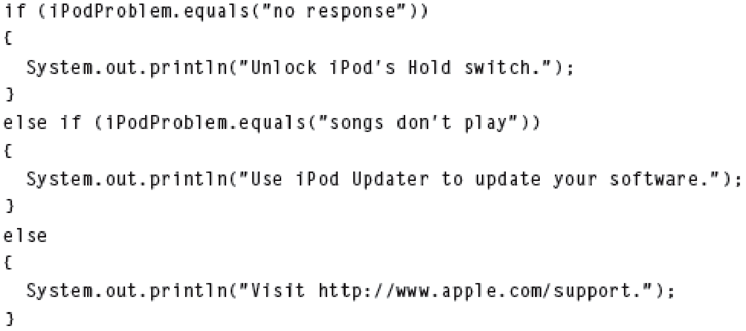
```
if (if-condition)
{
  if-statement(s)
}
else if (else-if-condition)
{
  else-if-statement(s)
}
.
.
.
else
{
  else-statement(s)
}
```

optional additional else if's
optional
if-condition
true
statement(s)
false
else-if-condition
true
else-if-statement(s)
false
else-statement(s)
following-statement

Figure 4.3 Syntax and semantics for the "if, else if" form of the if statement

Take several minutes and examine Figures 4.1, 4.2, and 4.3. The figures show the syntax and semantics for the three forms of the Java if statement. The *semantics* of a statement is a description of how the statement works. For example, Figure 4.1's flowchart illustrates the semantics of the "if" form of the if statement by showing the flow of control for different values of the if statement's condition.

Most of what you see in the if statement figures should look familiar because it parallels what you learned in Chapter 2. But the "if, else if" form of the if statement deserves some extra attention. You may include as many "else if" blocks as you like—more "else if" blocks for more choices. Note that the "else" block is optional. If all the conditions are false and there's no "else" block, none of the statement blocks is executed. Here's a code fragment that uses the “if, else if” form of the if statement to troubleshoot iPod1 problems:



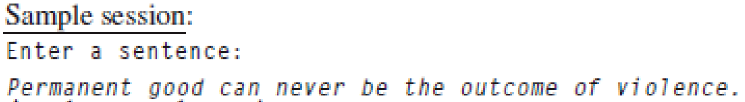
```
if (iPodProblem.equals("no response"))
{
  System.out.println("Unlock iPod's Hold switch.");
}
else if (iPodProblem.equals("songs don't play"))
{
  System.out.println("Use iPod Updater to update your software.");
}
else
{
  System.out.println("Visit http://www.apple.com/support.");
}
```

Practice Problem

Use design output to specify the problem.

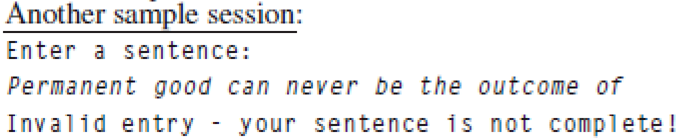
Page 124

Now let’s put what you’ve learned into practice by using the if statement within a complete program. Suppose you’re asked to write a sentence-tester program that checks whether a user-entered line ends with a period. Your program should print an error message if the last character in the line is not a period. In writing the program, use a sample session as a guide. Note that the italicized Mahatma Gandhi quote is a user-entered input value.



Sample session:

```
Enter a sentence:
Permanent good can never be the outcome of violence.
```



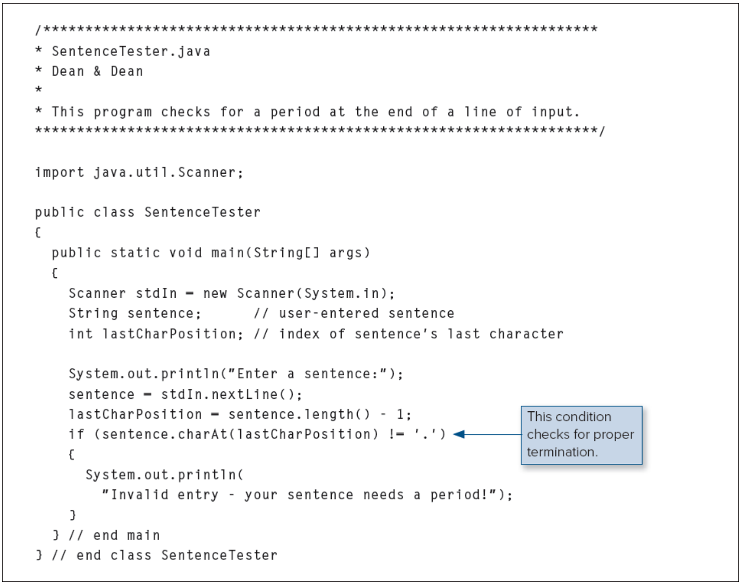
Another sample session:

```
Enter a sentence:
Permanent good can never be the outcome of
Invalid entry - your sentence is not complete!
```

As your first step in implementing a solution, use pseudocode to generate an informal outline of the basic logic:

```
print “Enter a sentence: ”
input sentence
if sentence’s last character is not equal to ‘.’
    print “Invalid entry – your sentence is not complete!”
```

Note the simple “if” form of the if statement. That’s appropriate because there’s a need to do something (print an invalid entry message) or nothing. Why nothing? Because the problem description does not say to print anything for user entries that are legitimate sentences. In other words, the program should skip what’s in the if statement if you finish the sentence properly. Now we suggest that you try writing the Java code to implement this algorithm. You’ll need to use a couple of the String methods described near the end of Chapter 3. When you’re ready, look at the SentenceTester solution in Figure 4.4.



```
/*******************************************************************
* SentenceTester.java
* Dean & Dean
*
* This program checks for a period at the end of a line of input.
*******************************************************************/

import java.util.Scanner;

public class SentenceTester
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String sentence;      // user-entered sentence
    int lastCharPosition; // index of sentence's last character

    System.out.println(”Enter a sentence:”);
    sentence = stdIn.nextLine();
    lastCharPosition = sentence.length() - 1;
    if (sentence.charAt(lastCharPosition) != '.')
    {
      System.out.println(
        ”Invalid entry - your sentence needs a period!”);
    }
  } // end main
} // end class SentenceTester
```

This condition checks for proper termination.

Figure 4.4 SentenceTester program

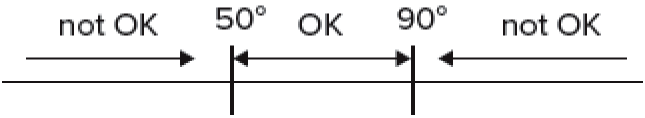
How does the SentenceTester program determine whether the last character is a period? Suppose the user enters “Hello.” In that case, what value would be assigned to the lastCharPosition variable? String's length method returns the number of characters in a string. The number of characters in "Hello." is six. Because the first position is zero, lastCharPosition would get assigned a value of 5 (the first character is 0, then 1, 2, 3, 4, 5 for the next characters). The purpose of finding the last character's index position value is to see if the last character in the user-entered value is a period. To see if it is, we use lastCharPosition as the argument in a charAt method call. String's charAt method returns the character at a specified index position within a string. The index position of the period in "Hello." is 5, and the if condition checks whether the user-entered value's last character is a period.

4.4 && Logical Operator

Up to this point, all of our if statement examples have used simple conditions. A simple condition evaluates directly to either true or false. In the next three sections, we introduce you to logical operators, like the “and” operator (&&) and the “or” operator (||), which make it possible to construct *compound* conditions. A compound condition is a conjunction (either an “anding” or an “oring”) of two or more conditions. When you have a compound condition, each part of the compound condition evaluates to either true or false, and then the parts combine to produce a composite true or false for the whole compound condition. The combining rules are what you might expect: When you “and” two conditions together, the combination is true only if the first condition is true and the second condition is true. When you “or” two conditions together, the combination is true if the first condition is true or the second condition is true. You’ll see plenty of examples as the chapter progresses.

&& Operator Example

Let’s begin our discussion of logical operators with an example that uses the && operator. (Note: is pronounced “and”). Suppose you want to print “OK” if the temperature is between 50 degrees and 90 degrees and print “not OK” otherwise:



Here’s a pseudocode description of the problem:

```
if temp $\geq 50$ and $\leq 90$
    print “OK”
else
    print “not OK”
```

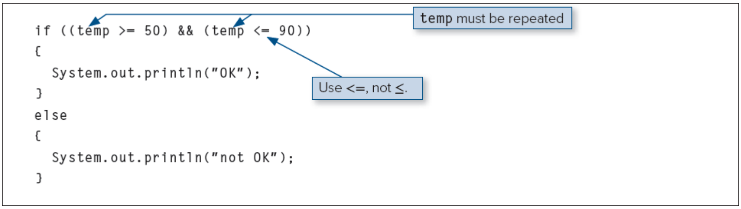
Think about where boundary values go.

Notice that the pseudocode condition

uses ≥ and ≤ rather than > and <. The original problem specification says to print "OK" if the temperature is between 50 degrees and 90 degrees. When people say "between," they usually, but not always, mean to include the end points. Thus, we assumed that the 50 and 90 end points were supposed to be included in the OK range, and we chose to use ≥ and ≤ accordingly. But in general, if you're writing a program and you're unsure about the end points for a particular range, you should not assume. Instead, you should ask the customer what he or she wants. The end points are important.

See Figure 4.5. It shows the Java implementation for the temperature-between-50-and-90 problem. In Java, if both of two criteria must be met for a condition to be satisfied (e.g., temp $>=$ 50 and temp $<=$ 90), then separate the two criteria with the &&

operator. As indicated by Figure 4.5’s first callout, if both criteria use the same variable (e.g., temp), you must include the variable on both sides of the &&. Note the use of > = and < =. In pseudocode, it’s OK to use ≥, ≤, or even the words “greater than or equal to,” and “less than or equal to.” But in Java, you must use > = and < =.



```
if ((temp >= 50) && (temp <= 90))
{
  System.out.println(”OK”);
}
else
{
  System.out.println(”not OK”);
}
```

`temp` must be repeated

Use <=, not ≤.

Figure 4.5 Java implementation of the temperature-between-50-and-90 problem

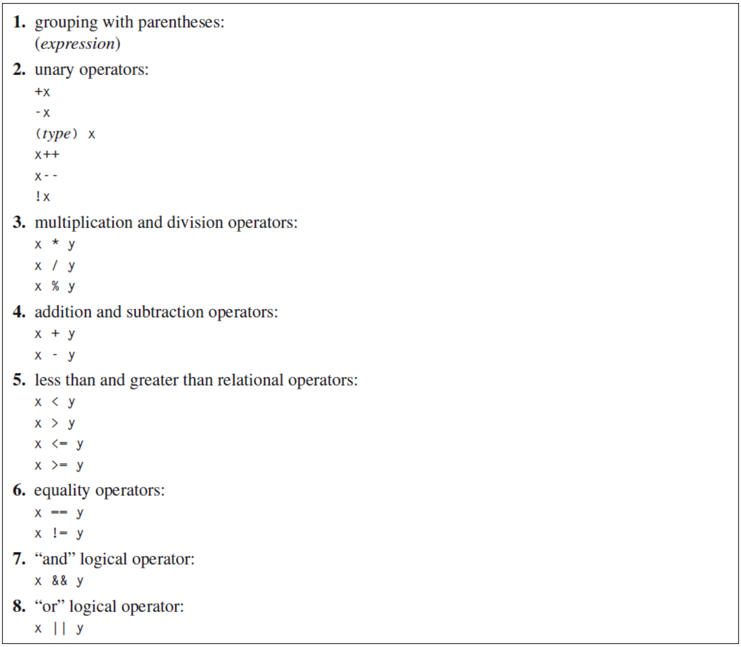
Operator Precedence

In Figure 4.5, note the parentheses around each of the two temperature comparisons. They force evaluation of the comparisons before evaluation of the &&. What would happen if we omitted those inner parentheses? To answer that sort of question, you need to refer to an operator precedence table. Appendix 2 provides a complete operator precedence table, but most of the cases you'll encounter are covered by the abbreviated precedence table in Figure 4.6. All operators within a particular numbered group have equal precedence, but operators at the top of the figure (in groups 1, 2, . . .) have higher precedence than operators at the bottom of the figure (in groups . . . 7, 8).

Figure 4.6 shows that the comparison operators > = and < = have higher precedence than the logical operator &&. Thus, the > = and < = operations execute before the operation—even if the inner parentheses in the condition in Figure 4.5 are omitted. In other words, we could have written Figure 4.5's condition more simply, like this:

```
if (temp >= 50 && temp <= 90)
```

You may include these extra parentheses or not, as you wish. We included them in Figure 4.5 to emphasize the order of evaluation in this initial presentation, but in the future, we will often omit them to minimize clutter.



1. grouping with parentheses:
(*expression*)

2. unary operators:
`+x`
`-x`
(*type*) `x`
`x++`
`x--`
`!x`

3. multiplication and division operators:
`x * y`
`x / y`
`x % y`

4. addition and subtraction operators:
`x + y`
`x - y`

5. less than and greater than relational operators:
`x < y`
`x > y`
`x <= y`
`x >= y`

6. equality operators:
`x == y`
`x != y`

7. "and" logical operator:
`x && y`

8. "or" logical operator:
`x || y`

Figure 4.6 Abbreviated operator precedence table (see Appendix 2 for complete table) The operator groups at the top of the table have higher

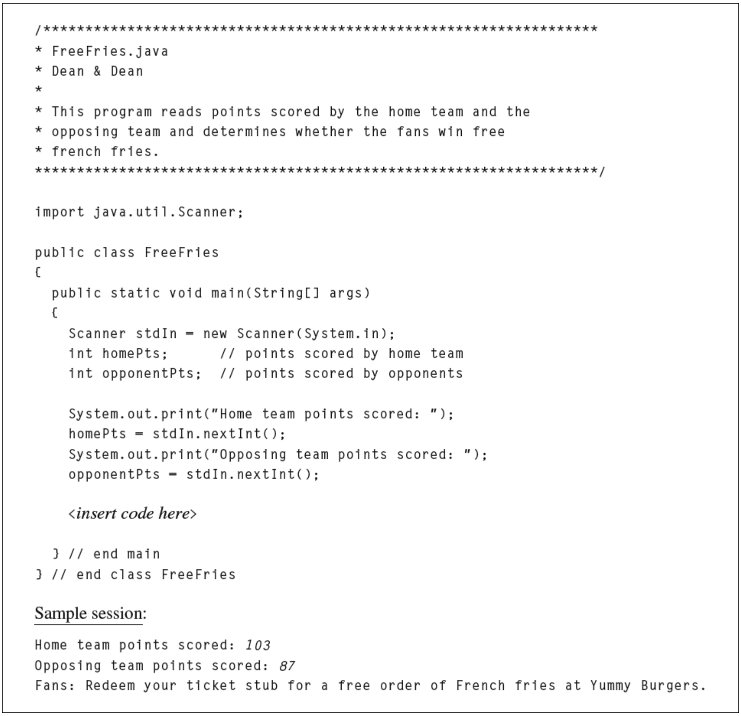
precedence than the operator groups at the bottom of the table. All operators within a particular group have equal precedence. If an expression has two or more same-precedence operators, then within that expression, operators on the left execute before those on the right.

Another Example

For another example, consider commercial promotions at sports events. Suppose the local Yummy - Burgers restaurant is willing to provide free French fries to all fans at a basketball game whenever the home team wins and scores at least 100 points. The problem is to write a program that prints the following message whenever that condition is satisfied:

“Fans: Redeem your ticket stub for a free order of French fries at Yummy Burgers.”

Figure 4.7 shows the framework. Within the figure, note where it says <*insert code here*>. Before looking ahead at the answer, see if you can provide the inserted code on your own.



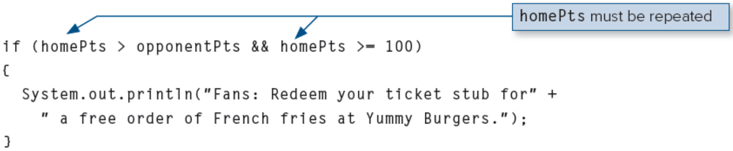
```
/*********************************************************************
* FreeFries.java
* Dean & Dean
*
* This program reads points scored by the home team and the
* opposing team and determines whether the fans win free
* french fries.
*********************************************************************/

import java.util.Scanner;

public class FreeFries
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int homePts;      // points scored by home team
    int opponentPts;  // points scored by opponents

    System.out.print("Home team points scored: ");
    homePts = stdIn.nextInt();
    System.out.print("Opposing team points scored: ");
    opponentPts = stdIn.nextInt();

    <insert code here>

  } // end main
} // end class FreeFries
```

Sample session:

```
Home team points scored: 103
Opposing team points scored: 87
Fans: Redeem your ticket stub for a free order of French fries at Yummy Burgers.
```

Figure 4.7 FreeFries program with "and" condition

Here's what you should insert:

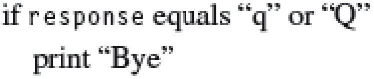


homePts must be repeated

```
if (homePts > opponentPts && homePts >= 100)
{
  System.out.println("Fans: Redeem your ticket stub for" +
    " a free order of French fries at Yummy Burgers.");
}
```

4.5 || Logical Operator

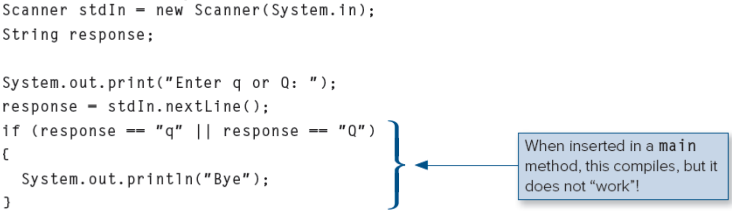
Now let’s look at the complement to the “and” operator—the “or” operator. Assume that you have a variable named response that contains (1) a lowercase or uppercase “q” if the user wants to quit or (2) some other character if the user wants to continue. Write a code fragment that prints “Bye” if the user enters either a lowercase or uppercase “q.” Using pseudocode, you’d probably come up with something like this for the critical part of the algorithm:



```
if response equals “q” or “Q”
   print “Bye”
```

Note the “or” in the if statement’s condition. That works fine for pseudocode, where syntax rules are lenient, but for Java, you must use || for the “or” operation, not “or.” (Note: || is pronounced “or.”) To enter the ||

operator on your computer, look for the vertical bar key on your keyboard and press it twice. Here's a tentative Java implementation of the desired code fragment:



```
Scanner stdIn = new Scanner(System.in);
String response;

System.out.print("Enter q or Q: ");
response = stdIn.nextLine();
if (response == "q" || response == "Q")
{
  System.out.println("Bye");
}
```

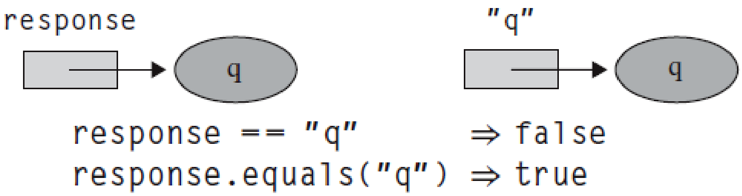
When inserted in a `main` method, this compiles, but it does not "work"!

Note that the response variable appears twice in the if statement's condition. That's necessary because if both sides of an || condition involve the same variable, you must repeat the variable.

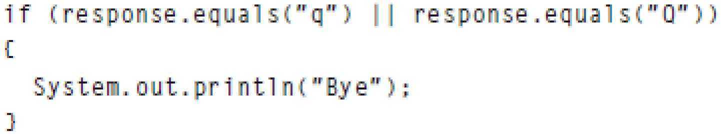
The callout indicates that something is wrong. What is it? If you insert this code fragment into a valid program shell, the program compiles and runs. But when a user responds to the prompt by dutifully entering either "q" or "Q," nothing happens. The program does not print "Bye." Why not? Should we have used interior parentheses in the “if” condition? Figure 4.6 shows that the == operator has a higher precedence than the || operator, so what we did was OK. The problem is something else.

Don’t Use == to Compare Strings

⚠ The problem is with the response == "q" and response == "Q" expressions. We’ll focus on the response == "q" expression. The response string variable and the “q” string literal both hold memory addresses that point to string objects; they don’t hold string objects themselves. So when you use ==, you’re comparing the memory addresses stored in the response string variable and the “q” string literal. If the response string variable and the “q” string literal contain different memory addresses (i.e., they point to different string objects), then the comparison evaluates to false, even if both string objects contain the same sequence of characters. The following picture shows what we're talking about. The arrows represent memory addresses. Because they point to two different objects, response == "q" evaluates to false.

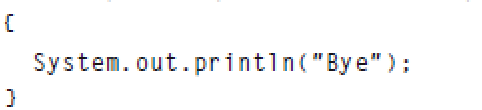


So what can you do to solve this problem? In Chapter 3, you learned to use the equals method to test strings for equality. The equals method compares the string objects pointed to by the memory addresses. In the above picture, the string objects hold the same sequence of characters, q and q, so the method call, response.equals("q"), returns true, which is what you want. Here's the corrected code fragment:



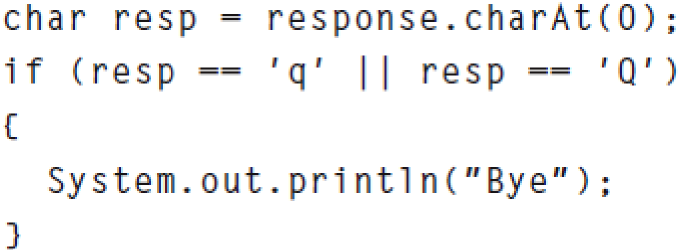
```
if (response.equals("q") || response.equals("Q"))
{
  System.out.println("Bye");
}
```

Or, as a more compact alternative, use the equalsIgnoreCase method like this:



```
if (response.equalsIgnoreCase("q"))
{
  System.out.println("Bye");
}
```

A third alternative is to use the String class’s charAt method to convert the string input into a character and then use the == operator to compare that character with the character literals, ‘q’ and ‘Q’:



```
char resp = response.charAt(0);
if (resp == 'q' || resp == 'Q')
{
  System.out.println("Bye");
}
```

The devil is in the details.

These implementations are not trivial translations from the pseudocode that specified the algorithm. It’s important to organize your thoughts before you start writing Java code. But even very good preparation does not eliminate the need to keep thinking as you proceed. Details matter also!

Errors

We made a big deal about not using == to compare strings because it's a very easy mistake to make and it's a hard mistake to catch. It's easy to make this mistake because you use == all the time when comparing primitive values. It's hard to catch this mistake because programs that use == for string comparison compile and run with no reported errors. No reported errors? Then why worry? Because although there are no reported errors, there are still errors—they're called logic errors.

Be careful. Test every aspect.

A *logic error* manifests itself when your program runs to completion without an error message, and the output is wrong. Logic errors are the hardest errors to find and fix because there's no error message glaring at you, telling you what you did wrong. To make matters worse, using == for string comparison generates a logic error only some of the time, not all the time. Because the logic error occurs only some of the time, programmers can be lulled into a false sense of confidence that their code is OK, when in reality it's not OK.

⚠ There are three main categories of errors—compile-time errors, runtime errors, and logic errors. A compile-time error is an error that is identified by the compiler during the compilation process. A runtime error is an error that occurs while a program is running and it causes the program to terminate abnormally. The compiler generates an error message for a compile-time error, and the Java virtual machine (JVM) generates an error message for a runtime error. Unfortunately, there are no error messages for a logic error. It’s up to the programmer to fix logic errors by analyzing the output and thinking carefully about the code.

4.6 ! Logical Operator

Now it’s time to consider the logical “not” operator (!). Assume that you have a char variable named resp that contains (1) a lowercase or uppercase ‘q’ if the user wants to quit or (2) some other character if the user wants to continue. This time, the goal is to print “Let’s get started. . . .” if resp contains anything other than a lowercase or uppercase “q”. You could use an “if, else” statement with an empty “if” block like this:

```
if (resp == 'q' || resp == 'Q')
{ }
else
{
  System.out.println("Let's get started. . . .");
  . . .
```

But this is not very elegant. Programmers often use the term *elegant* to describe code that is well written and has “beauty.” More specifically, elegant code is easy to understand, easy to update, robust, reasonably compact, and efficient. The above code’s empty “if” block is inelegant because it’s not compact. If you ever have an empty “if” block with a nonempty “else” block, you should try to rewrite it as just an “if” block with no “else” block. The trick is to invert the if statement’s condition. In the above example, that means testing for the absence of lowercase or uppercase ‘q’ rather than the presence of lowercase or uppercase ‘q.’ To test for the absence of lowercase or uppercase ‘q,’ use the ! operator.

The ! operator changes true values into false values and vice versa. This true-to-false, false-to-true toggling functionality is referred to as a “not” operation, and that’s why the ! operator is called the “not” operator. Because we want to print “Let’s get started. . . .” if the above if statement’s condition is not true, we insert ! at the left of the condition like this:

```
if (!(resp == 'q' || resp == 'Q'))
{
  System.out.println("Let's get started. . . .");
  . . .
```

Note that the ! is inside one set of parentheses and outside another set. Both sets of parentheses are required. The outer parentheses are necessary because the compiler requires parentheses around the entire condition. The inner parentheses are also necessary because without them, the ! operator would operate on the resp variable instead of on the entire condition. Why? Because the operator precedence table (Figure 4.6) shows that the ! operator has higher precedence than the == and || operators. The way to force the == and || operators to be executed first is to put them inside parentheses.

Don’t confuse the ! (not) operator with the != (inequality) operator. The ! operator returns the opposite value of the given expression (a true expression returns false and a false expression returns true). The != operator asks a question—are the two expressions unequal?

4.7 Switching Constructs

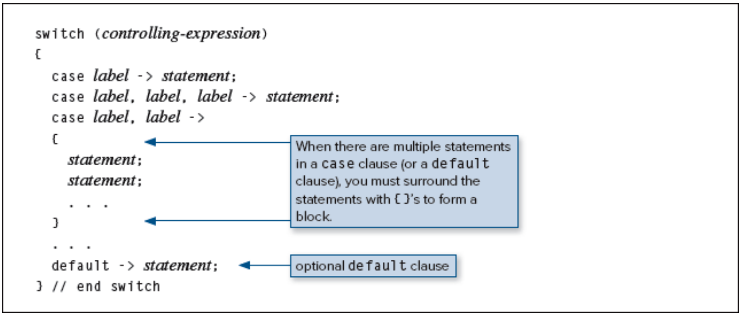
Like the “if, else if” form of the if statement, a switch construct allows you to follow one of several paths. But a key difference between the switch construct and an “if, else if” statement is that a switch uses just a single expression to determine which path to take. (With an “if, else if” statement, the determination of which path to take is based on multiple expressions, one for each path.) Using a single expression can lead to a more compact, more understandable implementation. Think of driving on Route 1 along the California coastline and coming to a junction with alternate routes through and around a city. The different routes are better at certain times of the day. If it’s 8 am or 5 pm, you should take the outer business loop to avoid rush-hour traffic. If it’s 8 pm, you should take the coastal bluffs route to appreciate the scenic sunset view. If it’s any other time, you should take the through-the-city route because it is the most direct and fastest. Using the value of a single controlling expression, time of day, to determine the route is like using a switch construct.

azgek/iStock/Getty Images

There are two types of switch constructs—a switch statement and a switch expression. As a statement, the switch construct is a stand-alone entity. As an expression, the switch construct cannot stand by itself; it must be embedded within a larger statement. We'll describe the switch statement first, and then move on to the switch expression.

switch Statement

Take a look at the switch statement's syntax in Figure 4.8a. When executing a switch statement, the JVM evaluates the controlling expression (which might be a variable, a mathematical expression, or a method call that returns a value), and it attempts to match the expression’s value with one of the case labels. A *case label* is a constant. For example, you can use integers (e.g., 2, 3, 4), or characters (e.g., ‘y’, ‘Y’, ‘n’, ‘N’) for your case labels. After finding a matching label, the JVM jumps to the statement that follows the case label’s arrow (->), executes that statement, and then jumps below the switch statement’s closing brace. If there’s no case label that matches the controlling expression’s value, then control jumps to the default clause (if there is a default clause) or below the switch statement if there is no default clause.



```
switch (controlling-expression)
{
  case label -> statement;
  case label, label, label -> statement;
  case label, label ->
  {
    statement;
    statement;
    . . .
  }
  . . .
  default -> statement;
} // end switch
```

When there are multiple statements in a `case` clause (or a `default` clause), you must surround the statements with { }’s to form a block.

optional `default` clause

Figure 4.8a switch statement syntax

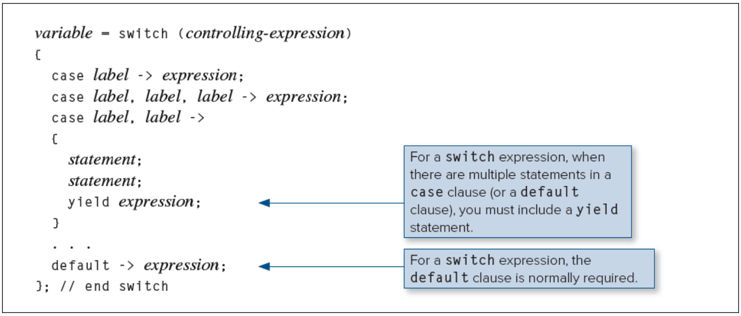
A switch statement consists of a switch header (the reserved word switch followed by a parentheses-enclosed controlling expression) and then a switch block (a pair of braces that surround the main part of the switch statement). The switch block contains a set of case clauses. Each case clause begins with the reserved word, case, followed by a label or a list of comma-separated labels, then an arrow (->), and then finally a statement. Each label must be a constant, typically an int, char, String, or an enumerated type.2 Optionally, you may add a default clause, as shown in Figure 4.8a, which gets executed if the JVM finds that the controlling expression's value does not match any of the case labels.

If you need to execute multiple statements in a case clause or a default clause, then you must enclose those statements in braces to form a compound statement (also known as a block). That should sound familiar because with an if statement, you must use braces when there are multiple statements inside the if part or the else part.

switch Expression

Now let's move on to the switch expression. Most of its syntax and semantics are identical to the switch statement. Thus, much of Figure 4.8b's switch expression should look familiar. One difference is that for each case clause, after the arrow, there's an expression. When the JVM executes the switch expression, it looks for a matching case label within a case clause, and it returns the expression that follows the case clause's arrow.

In Figure 4.8b, note that the switch expression is embedded in an assignment statement. Note the *variable* = code at the top. And note the semicolon at the bottom, next to the switch expression’s closing brace. Those things are required as part of the standard syntax for an assignment statement. You can use a switch expression anywhere there’s a need for a value whose type matches the type returned by the switch expression. For example if you want to print one of several different strings, you can embed a switch expression in a print statement and have the switch expression return a string. In that case, you’d start with System.out.println. You’d still end with a semicolon because that’s required for a print statement.



```
variable = switch (controlling-expression)
{
  case label -> expression;
  case label, label, label -> expression;
  case label, label ->
  {
    statement;
    statement;
    yield expression;
  }
  . . .
  default -> expression;
}; // end switch
```

For a `switch` expression, when there are multiple statements in a `case` clause (or a `default` clause), you must include a `yield` statement.

For a `switch` expression, the `default` clause is normally required.

Figure 4.8b Syntax for a switch expression within an assignment

statement

You might recall that with a switch statement, the default clause is optional. With a switch expression, it’s normally required. The only exception is if you use an enumerated type (described in Chapter 12) for the controlling expression’s value and you include all possible controlling expression values in the switch expression’s set of case labels. For that scenario, you can omit the default clause. But for a controlling expression of any other type, a default clause is required. Without it, there’s no way to satisfy the compiler that the switch expression’s set of case labels covers all of the possible values generated by the controlling expression. Here’s the compilation error when the default clause is missing:

```
error: the switch expression does not cover all possible input values
```

In Figure 4.8b, note the third case

clause that contains multiple statements surrounded by braces. If you have multiple statements in a case clause or a default clause, you're required to surround them with braces. That's the same as with the switch statement. For a switch expression, if you have multiple statements in a case clause or a default clause, you're required to include a yield statement as the last statement. The yield statement's expression is the value returned by the switch expression. The yield statement isn't used much—never for the switch statement, and for the switch expression only when there are multiple statements for a case clause or default clause.

Referring to Figures 4.8a and 4.8b, take note of these details, which can be easily overlooked:

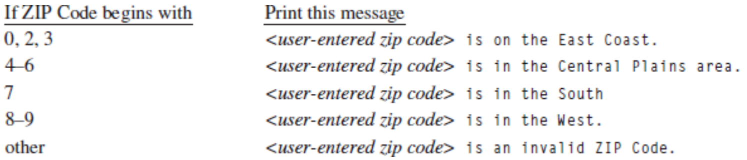
- There must be parentheses around the controlling expression.
- The controlling expression may

evaluate to a limited set of types. It’s illegal to use a boolean, a long, or a floating-point type.

- Although it’s common for the controlling expression to be just a single value, it can be an expression with multiple parameters, operators, and even method calls like stdIn.next(), provided that the expression evaluates to one of the permitted types.
- It’s good style to include // end switch after the switch construct’s closing brace.

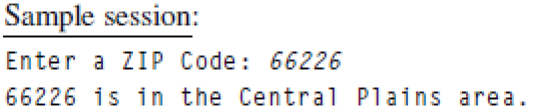
ZIP Code Program

For practice with switching, let’s look at a program that reads in a ZIP Code and uses the first digit to print the associated geographic area. Here’s what we’re talking about:



If ZIP Code begins with	Print this message
0, 2, 3	*<user-entered zip code>* `is on the East Coast.`
4–6	*<user-entered zip code>* `is in the Central Plains area.`
7	*<user-entered zip code>* `is in the South`
8–9	*<user-entered zip code>* `is in the West.`
other	*<user-entered zip code>* `is an invalid ZIP Code.`

The first digit of a U.S. postal ZIP Code identifies a particular geographic area within the United States. ZIP Codes that start with 0, 2, or 3 are in the east, ZIP Codes that start with 4, 5, or 6 are in the central region, and so on.[3] Our program should prompt the user for his or her ZIP Code and use the first character of the entered value to print the user's geographical region. In addition to printing the geographical region, our program should echo print the user's ZIP Code. (Echo print means print out an input exactly as it was read in.) Here's an example of what the program should do:



Sample session:

```
Enter a ZIP Code: 66226
66226 is in the Central Plains area.
```

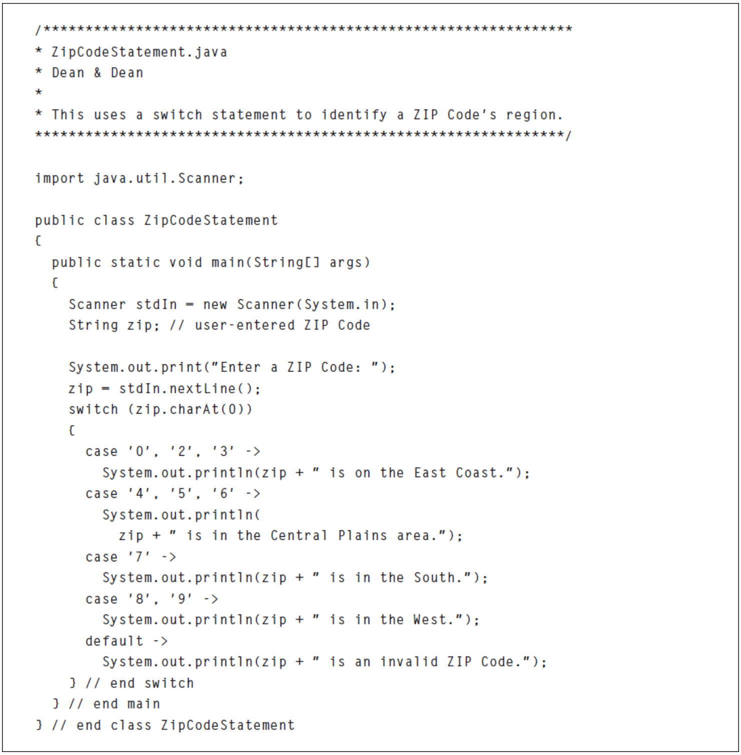
Figure 4.9a shows the ZipCodeStatement program, which uses a switch statement. Look at the controlling expression, zip.charAt(0), which evaluates to the first character in zip. As an alternative, we could have started by reading the first character into a separate variable (e.g., firstChar), and then inserted that variable into the controlling expression. But because we need the first character only once, we make the code more compact by embedding zip.charAt(0) directly in the controlling expression's parentheses.

The switch statement compares the controlling expression's character with each of the case labels until it finds a match. Because the controlling expression's charAt method returns a char value, the case labels must all be of type char. Therefore, the case labels must have single quotes around them. If you use double quotes or no quotes, you'll get a compilation error.

Did you notice all the print statements in the ZipCodeStatement program? Let’s create a second version of the program that eliminates that print clutter. Rather than including a print statement in every case clause of the program’s switch statement, we print the ZIP Code and its region with just one print statement at the bottom. See the ZipCodeExpression program in Figure 4.9b for details.

In the ZipCodeExpression program, we declare a String variable named zipRegion, which holds the region for the user-entered ZIP Code. We replace the switch statement with a switch expression, and the switch expression’s case clauses return a message about the ZIP Code’s region. For example, here’s the first such case clause:

```
case '0', '2', '3' -> "on the East Coast";
```



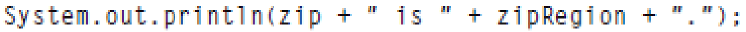
```
/*******************************************************************
* ZipCodeStatement.java
* Dean & Dean
*
* This uses a switch statement to identify a ZIP Code’s region.
*******************************************************************/

import java.util.Scanner;

public class ZipCodeStatement
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String zip; // user-entered ZIP Code

    System.out.print(”Enter a ZIP Code: ”);
    zip = stdIn.nextLine();
    switch (zip.charAt(0))
    {
      case ’0’, ’2’, ’3’ ->
        System.out.println(zip + ” is on the East Coast.”);
      case ’4’, ’5’, ’6’ ->
        System.out.println(
          zip + ” is in the Central Plains area.”);
      case ’7’ ->
        System.out.println(zip + ” is in the South.”);
      case ’8’, ’9’ ->
        System.out.println(zip + ” is in the West.”);
      default ->
        System.out.println(zip + ” is an invalid ZIP Code.”);
    } // end switch
  } // end main
} // end class ZipCodeStatement
```

Figure 4.9a Using a switch statement in the ZipCodeStatement program

Note the first line of the switch expression:

```
zipRegion = switch (zip.charAt(0))
```

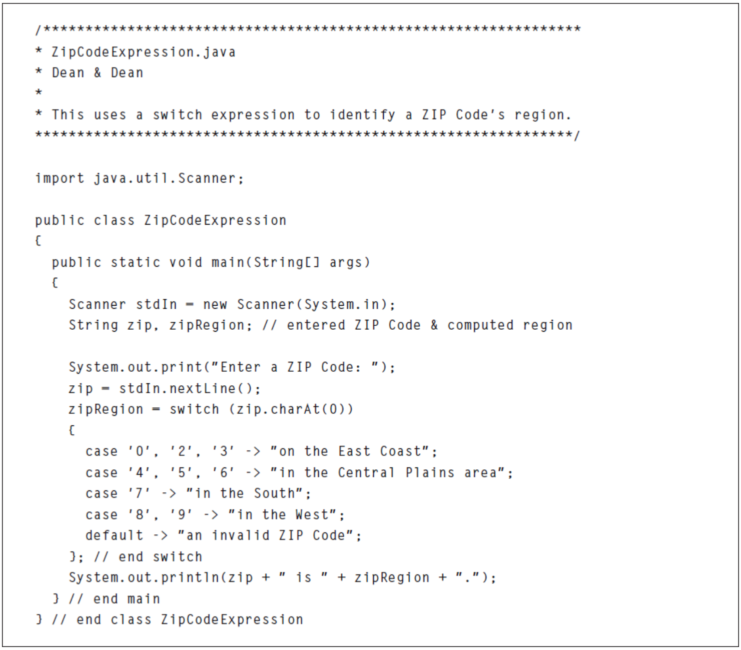
Thus, the switch expression’s returned value gets assigned into the zipRegion variable. And finally, below the switch statement, we add this print statement:



```
System.out.println(zip + " is " + zipRegion + ".");
```

Page 137

In general, statement switching is more appropriate when the alternatives are actions, and expression switching is more appropriate when the alternatives are items. In this section’s introductory example of alternate highway routes, if what matters most is steering action, statement switching would be more appropriate. But if what matters most is identification of a highway number, expression switching would be more appropriate. For a program that prints a ZIP Code and its associated geographical region, expression switching is more appropriate because the alternatives are items (strings), not actions.



```
/*******************************************************************
* ZipCodeExpression.java
* Dean & Dean
*
* This uses a switch expression to identify a ZIP Code's region.
*******************************************************************/

import java.util.Scanner;

public class ZipCodeExpression
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String zip, zipRegion; // entered ZIP Code & computed region

    System.out.print("Enter a ZIP Code: ");
    zip = stdIn.nextLine();
    zipRegion = switch (zip.charAt(0))
    {
      case '0', '2', '3' -> "on the East Coast";
      case '4', '5', '6' -> "in the Central Plains area";
      case '7' -> "in the South";
      case '8', '9' -> "in the West";
      default -> "an invalid ZIP Code";
    }; // end switch
    System.out.println(zip + " is " + zipRegion + ".");
  } // end main
} // end class ZipCodeExpression
```

Figure 4.9b Using a switch expression in the ZipCodeExpression program

Traditonal switch Statement

The switching constructs described above were introduced with Java 12. There’s an older form of the switch statement, which you should be aware of because you’ll need to understand it when you see it in legacy code (code written for older programs), and some developers will continue to use the old switch statement for new programs.

The switching constructs described above use this syntax for their case clauses:

```
case label, label, label -> statement;
```

On the other hand, Java's traditional switch statement uses this syntax for its case clauses:

```
case label: case label: case label:
  statement(s);
  break;
```

Note that there's no braces surrounding the case clause's statements. Also note the break statement, which causes execution to jump below the switch statement and continue with whatever follows the switch statement. If there is no break statement for a case clause, after executing the case clause's statement(s), the JVM falls through to the next clause (a case clause or a default clause) and executes the next clause's statements regardless of whether there's a matching case label in the next clause. That *fall-through mechanism* enables programmers to avoid duplication of statements that perform identical operations in adjacent clauses. However, accidentally omitting a break statement generates a logical programming error, so be careful to normally include a break statement.

In addition to their propensity for error, break statements add annoying clutter. Therefore, for new programs, we recommend sticking with the switch expression and the newer-version switch statement. Nonetheless, because it'll be around for awhile, you should understand the traditional switch statement. Study the ZipCodeTraditionalSwitch program in Figure 4.9c, which uses a traditional switch statement.

Switching Constructs Versus "if,

else if” Form of the if Statement

As you now know, the switching constructs allow you to do one or more things from a list of multiple possibilities. But so does the “if, else if” form of the if statement, so why would you ever want to use a switching construct? Because switching provides a more elegant solution (cleaner, more compact organization) for certain kinds of problems.

But also consider the opposite question: Why would you ever want to use the “if, else if” form of the if statement rather than a switching construct? Because if statements are more flexible. With switching, each test (i.e., each case label) is limited to an exact match with one of the permitted types. With an if statement, each test can be a full-blooded expression, complete with operators, variables, and method calls.

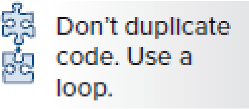
In a nutshell, when you need to do

one thing from a list of multiple possibilities:

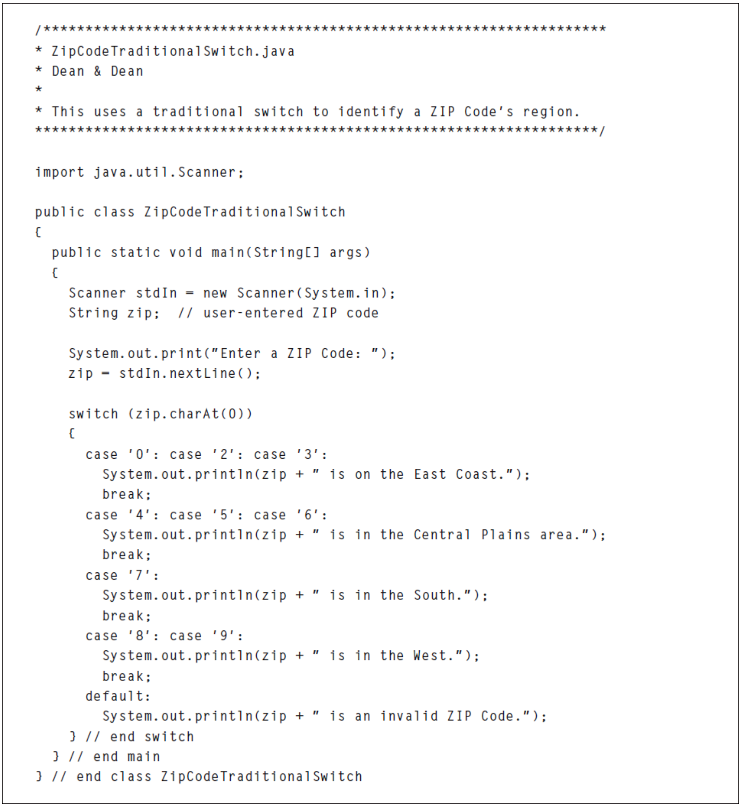
- Use switching if you need to match an int, char, String, or one of a limited set of values.
- Use an if statement if you need more flexibility.

4.8 while Loop

There are two basic categories of control statements—forward branching statements and looping statements. The if statement and the switching constructs implement *forward branching* functionality (so named because the decisions cause control to "branch" to a statement that is ahead of the current statement). The while loop, do loop, and for loop implement looping functionality. We describe the while loop in this section and the do and for loops in the next two sections. But first an overview of loops in general.


Don't duplicate code. Use a loop.

In solving a particular problem, one of the first and most important things to think about is whether there are any repetitive tasks. Repetitive tasks should normally be implemented with the help of a loop. For some problems, you can avoid a loop by implementing the repetitive tasks with consecutive sequential statements. For example, if you are asked to print “Happy Birthday!” 10 times, you could implement a solution with 10 consecutive print statements. But such a solution would be a poor one. A better solution is to insert a single print statement inside a loop that repeats 10 times. The loop implementation is better because it’s more compact. Also, updating is easier and safer because the updated code appears in only one place. For example, if you need to change “Happy Birthday!” to “Bon Anniversaire!” (“Happy Birthday!” in French), then it’s only a matter of changing one print statement inside a loop rather than updating 10 separate print statements.

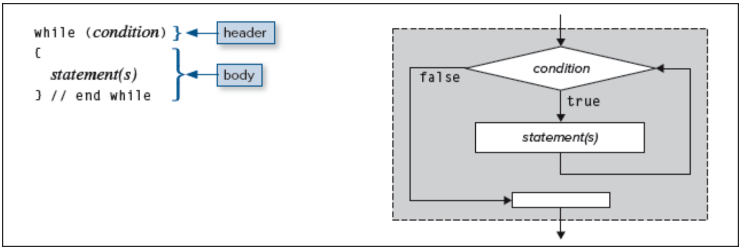


```
/*********************************************************************
* ZipCodeTraditionalSwitch.java
* Dean & Dean
*
* This uses a traditional switch to identify a ZIP Code's region.
*********************************************************************/

import java.util.Scanner;

public class ZipCodeTraditionalSwitch
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String zip;  // user-entered ZIP code

    System.out.print("Enter a ZIP Code: ");
    zip = stdIn.nextLine();

    switch (zip.charAt(0))
    {
      case '0': case '2': case '3':
        System.out.println(zip + " is on the East Coast.");
        break;
      case '4': case '5': case '6':
        System.out.println(zip + " is in the Central Plains area.");
        break;
      case '7':
        System.out.println(zip + " is in the South.");
        break;
      case '8': case '9':
        System.out.println(zip + " is in the West.");
        break;
      default:
        System.out.println(zip + " is an invalid ZIP Code.");
    } // end switch
  } // end main
} // end class ZipCodeTraditionalSwitch
```

Figure 4.9c Using a traditional switch statement in the ZipCodeTraditionalSwitch program

while Loop Syntax and Semantics

Now let's look at the simplest kind of loop, the while loop. Figure 4.10 shows the syntax and semantics of the while loop. The syntax for the while loop looks like the syntax for the if statement except that the word while is used instead of the word if. Don't forget the parentheses around the condition. Don't forget the braces, and don't forget to indent the subordinate statements they enclose.



```
while (condition)
{
  statement(s)
} // end while
```

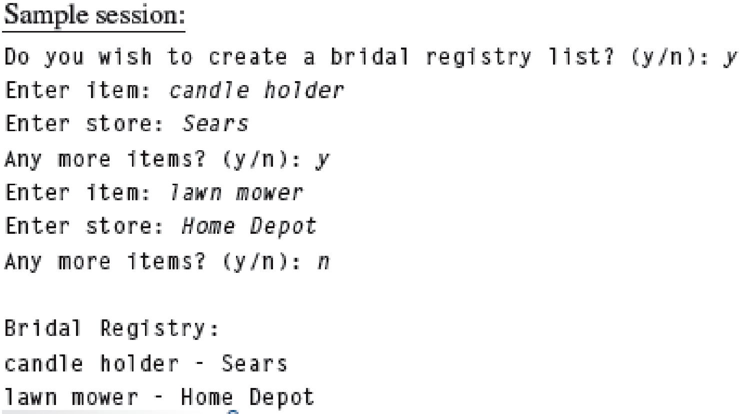
Figure 4.10 Syntax and semantics for the while loop

A while loop's condition is the same as an if statement's condition. It typically employs comparison and logical operators, and it evaluates to true or false. Here's how the while loop works:

1. Check the while loop's condition.
2. If the condition is true, execute the while loop's body (the statements that are inside the braces), jump back to the while loop's condition, and repeat step 1.
3. If the condition is false, jump to below the while loop's body and continue with the next statement.

Example

Now let's consider an example—a program that creates a bridal gift registry. More specifically, the program repeatedly prompts the user for two things—a gift item and the store where the gift can be purchased. When the user is done entering gift and store values, the program prints the bridal registry list. Study this sample session:

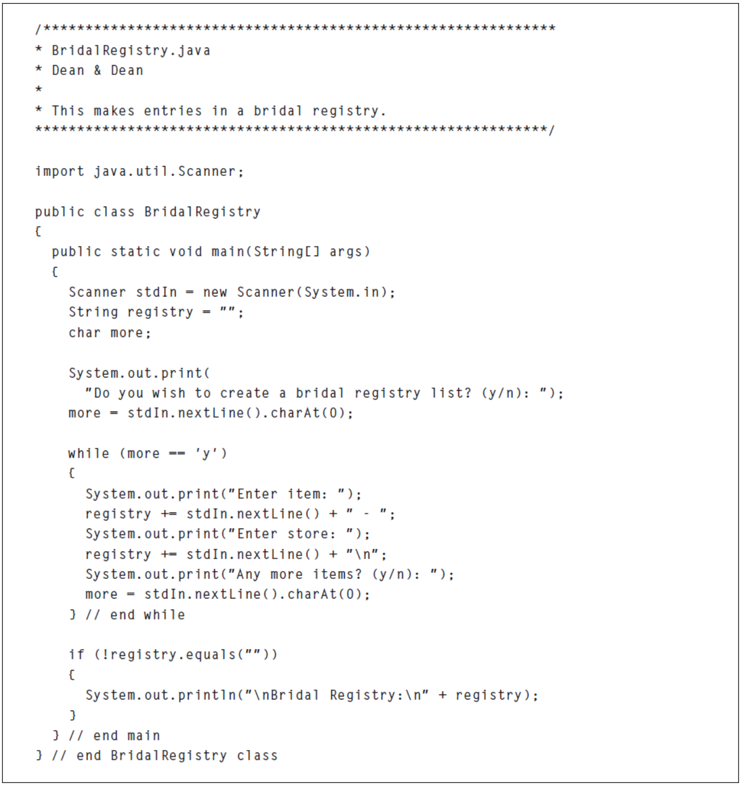


Sample session:

```
Do you wish to create a bridal registry list? (y/n): y
Enter item: candle holder
Enter store: Sears
Any more items? (y/n): y
Enter item: lawn mower
Enter store: Home Depot
Any more items? (y/n): n

Bridal Registry:
candle holder - Sears
lawn mower - Home Depot
```

Use I/O sample to specify the problem.

That's the problem specification. Our solution appears in Figure 4.11. As you can tell by the while loop's more == 'y' condition and the query at the bottom of the loop, the program employs a user-query loop. The initial query above the while loop makes it possible to quit without making any passes through the loop. If you want to force at least one pass through the loop, you should delete the initial query and initialize more like this:



```
/*************************************************************
* BridalRegistry.java
* Dean & Dean
*
* This makes entries in a bridal registry.
*************************************************************/

import java.util.Scanner;

public class BridalRegistry
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String registry = "";
    char more;

    System.out.print(
      "Do you wish to create a bridal registry list? (y/n): ");
    more = stdIn.nextLine().charAt(0);

    while (more == 'y')
    {
      System.out.print("Enter item: ");
      registry += stdIn.nextLine() + " - ";
      System.out.print("Enter store: ");
      registry += stdIn.nextLine() + "\n";
      System.out.print("Any more items? (y/n): ");
      more = stdIn.nextLine().charAt(0);
    } // end while

    if (!registry.equals(""))
    {
      System.out.println("\nBridal Registry:\n" + registry);
    }
  } // end main
} // end BridalRegistry class
```

Figure 4.11 BridalRegistry program with while loop and user-query termination

The BridalRegistry program illustrates several peripheral concepts that you'll want to remember for future programs. Within the while loop, note the += assignment statements, repeated here for your convenience:

```
registry += stdIn.nextLine() + " - ";
registry += stdIn.nextLine() + "\n";
```

The += operator comes in handy when you need to add incrementally to a string variable. The BridalRegistry program stores all the gift and store values in a single String variable named registry. Each new gift and store entry gets concatenated to the registry variable with the += operator.

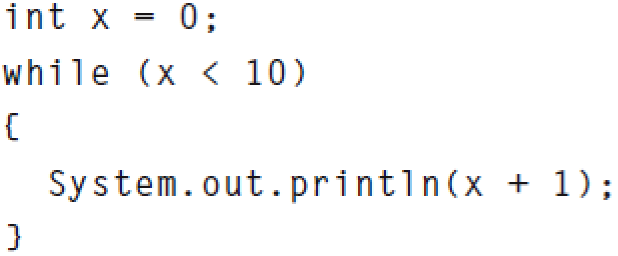
At the top and bottom of the BridalRegistry program's while loop, note the nextLine and charAt method calls, repeated here for your convenience:

```
more = stdIn.nextLine().charAt(0);
```

The method calls are *chained* together by inserting a dot between them. The nextLine() method call reads a line of input from the user and returns the input as a string. That string then calls the charAt(0), which returns the string's first character. Note that it's acceptable and fairly common to chain multiple method calls together like this.

Infinite Loops

Suppose you're trying to print the numbers 1 through 10. Will the following code fragment work?



```
int x = 0;
while (x < 10)
{
  System.out.println(x + 1);
}
```

The while loop body does just one thing—it prints 1 (because 0 + 1 is 1). It does not update x's value (because there's no assignment or increment statement for x). With no update for x, the while loop's condition (x < 10) always evaluates to true. That's an example of an *infinite loop*. The computer executes the statements in the loop body over and over—forever. When you have an infinite loop, the computer seems to freeze or "hang up."

Insert temporary print statements to see details.

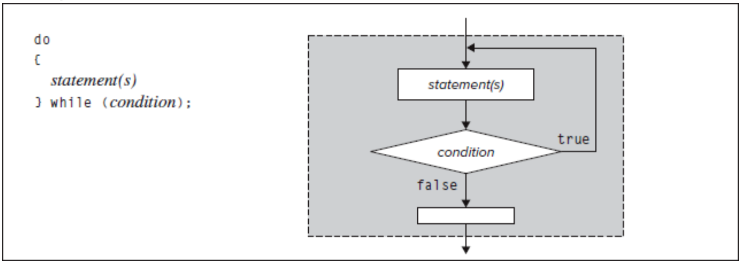
Sometimes what seems to be an infinite loop is just an extremely inefficient algorithm that takes a long time to finish. In either of these cases, you can figure out what's happening by inserting into the loop a diagnostic statement that prints a value you think should be changing in a certain way. Then run the program and watch what happens to that value.

4.9 do Loop

Now let's consider a second type of Java loop—the do loop. A do loop is appropriate when you're sure that you want the loop body to be executed at least one time. Because the do loop matches the way most computer hardware performs looping operations, it is slightly more efficient than the other types of loops. Unfortunately, its awkwardness makes it prone to programming error, and therefore some programmers don't like to use it. But at the very least, you need to be aware of it.

Syntax and Semantics

Figure 4.12 shows the do loop's syntax and semantics. Note that the do loop's condition is at the bottom. This contrasts with the while loop, where the condition is at the top. Having the condition tested at the bottom is how the do loop guarantees that the loop executes at least one time. Note the semicolon at the right of the condition. That's required by the compiler, and omitting it is a common error. Finally, note that the while part is on the same line as the closing brace—that's good style. It's possible to put while (*condition*); on the line after the closing brace, but that would be bad style because it would look like you're trying to start a new while loop.



```
do
{
  statement(s)
} while (condition);
```

Figure 4.12 Syntax and semantics for the do loop

Here's how the do loop works:

1. Execute the do loop's body.

2. Check the final condition.

3. If the condition is true, jump back to the top of the do loop and repeat step 1.

4. If the condition is false, continue with the statement immediately below the loop.

Practice Problem

Now let's illustrate the do loop with an example problem. Suppose you're asked to write a program that prompts the user to enter length and width dimensions for each room in a proposed house so that total floor space can be calculated for the entire house. After each length/width entry, ask the user if there are any more rooms. When there are no more rooms, print the total floor space.

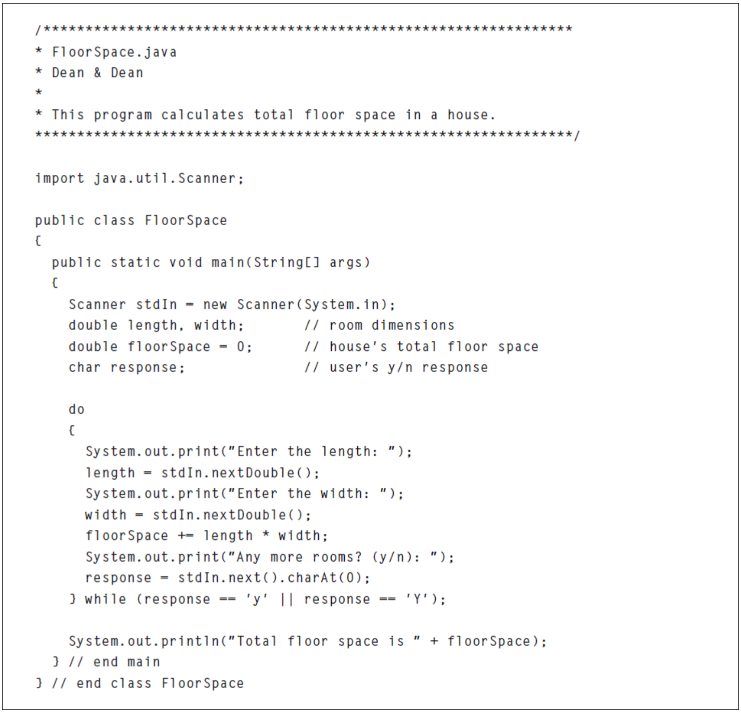
How many repeats?

To solve this problem, first ask whether a loop is appropriate. Does anything need to be repeated? Yes, you’ll want to read in dimensions repeatedly, so a loop is appropriate. To determine the type of loop, ask yourself: Will you always need to execute the read-in-the-dimensions loop body at least once? Yes, every house must have at least one room, so you’ll need to read in at least one set of dimensions. Thus, it’s appropriate to use a do loop for this problem. Now that you’ve thought through the looping issues, you're ready to put pencil to paper and write down your solution. Go for it.

When you're done working out a solution on your own, look at our solution in Figure 4.13. Did you prompt for length and width values within your do loop and then add the length times width product to a total floor space variable? Did you then prompt the user for a continue decision?

Compare the loop-termination technique used in the FloorSpace program with the loop-termination technique used in the BridalRegistry program in Figure 4.11. In the BridalRegistry program, we needed two user queries—one before the start of the loop and one within the loop just before its end. In the FloorSpace program, we need only one user query—within the loop just before its end. The do loop requires that there be at least one pass, but if this is acceptable, it requires fewer lines of code than the while loop.

Before leaving the FloorSpace program, take note of a style feature. Do you see the blank lines above and below the do loop? It’s good style to separate logical chunks of code with blank lines. Because a loop is a logical chunk of code, it’s nice to surround loops with blank lines unless the loop is very short (that is, less than about four lines).

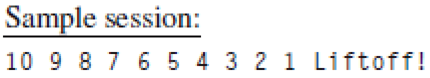


```
/*******************************************************************
* FloorSpace.java
* Dean & Dean
*
* This program calculates total floor space in a house.
*******************************************************************/

import java.util.Scanner;

public class FloorSpace
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double length, width;       // room dimensions
    double floorSpace = 0;      // house's total floor space
    char response;              // user's y/n response

    do
    {
      System.out.print(”Enter the length: ”);
      length = stdIn.nextDouble();
      System.out.print(”Enter the width: ”);
      width = stdIn.nextDouble();
      floorSpace += length * width;
      System.out.print(”Any more rooms? (y/n): ”);
      response = stdIn.next().charAt(0);
    } while (response == ’y’ || response == ’Y’);

    System.out.println(”Total floor space is ” + floorSpace);
  } // end main
} // end class FloorSpace
```

Figure 4.13 Using a do loop to calculate total floor space

4.10 for Loop

Now let’s consider a third type of loop—the for loop. A for loop is appropriate when you know the exact number of loop iterations before the loop begins. For example, suppose you want to perform a countdown from 10, like this:

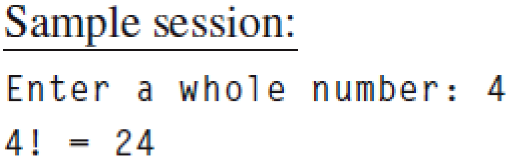


Sample session:

```
10 9 8 7 6 5 4 3 2 1 Liftoff!
```

In your program, you'll need to print 10 numbers, and you should print each number with the help of a print statement inside a loop. Because the print statement should execute 10 times, you know the exact number of iterations for the loop, 10. Therefore, you should use a for loop.

For another example, suppose you want to find the factorial of a user-entered number, like this:



Sample session:

```
Enter a whole number: 4
4! = 24
```

For 4 factorial, you need to multiply the values 1 through 4: 1 × 2 × 3 × 4 = 24. The three ×'s indicate that three multiplications are necessary. So 4 factorial requires three loop iterations. For the general case, where you need to find the factorial for a user-entered number, store the user-entered number in a count variable. Then multiply the values 1 through count like this:

$$1 \underbrace{*\ 2\ *\ 3\ *\ldots*}_{\texttt{count - 1}\text{ number of *'s}} \texttt{count}$$

The *'s indicate that count - 1 multiplications are necessary. So count factorial requires count - 1 loop iterations. Because you know the number of iterations for the loop (count - 1), use a for loop.

Syntax and Semantics

Figure 4.14 shows the for loop's syntax and semantics. The for loop header does a lot of work. So much work that it's split into three components—the *initialization*, *condition*, and *update* components. The following list explains how the for loop uses the three components. As you read the list, refer to Figure 4.14's flowchart to get a better idea of what’s going on.

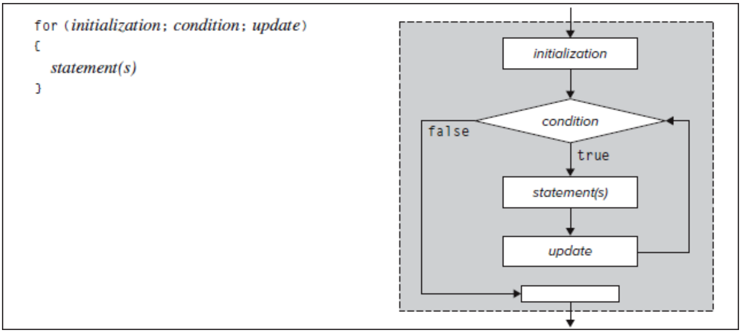
1. Initialization component

Before the first pass through the body of the loop, execute the initialization component.

2. Condition component

Before each loop iteration, evaluate the condition component:

- If the condition is true, execute the body of the loop.
- If the condition is false, terminate the loop (exit to the statement below the loop’s closing brace).



```
for (initialization; condition; update)
{
  statement(s)
}
```

initialization
condition
false
true
statement(s)
update

Figure 4.14 Syntax and semantics for the for loop

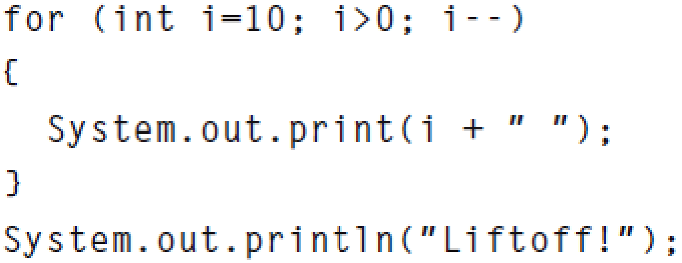
3. Update component

After each pass through the body of the loop, return to the loop header and execute the update component. Then, recheck the continuation condition in the second component, and if it's satisfied, go through the body of the loop again.

The header of a for loop is quite flexible. We discuss alternative configurations later in this book, in Section 12.12 of Chapter 12.

Countdown Example

Here is a code fragment for the countdown example mentioned at the start of this section:



```
for (int i=10; i>0; i--)
{
  System.out.print(i + " ");
}
System.out.println("Liftoff!");
```

Note that the same variable, i, appears in all three components of the for loop header. That variable is given a special name. It's called an *index variable.* Index variables in for loops are often, but not always, named i for "index." Even when it is not the letter i, a for-loop index is usually abbreviated as one letter or just a few letters. As a rule, we discourage highly abbreviated variable names. It's OK in this instance, however, because the context of the for-loop header clearly explains the index variable's meaning.

Index variables often start at a low value, increment up, and then stop when they reach a threshold set by the condition component. But in the above example, the index variable does just the opposite. It starts at a high value (10), decrements down, and then stops when it reaches the threshold of 0. Let's informally trace the example:

The initialization component

assigns 10 to the index, i.

The condition component asks “Is i > 0?” The answer is yes, so execute the body of the loop.

Print 10 (because i is 10), and append a space.

Because you’re at the bottom of the loop, the update component decrements i from 10 to 9.

The condition component asks “Is i > 0?” The answer is yes, so execute the body of the loop.

Print 9 (because i is 9) and append a space.

Because you’re at the bottom of the loop, the update component decrements i from 9 to 8.

The condition component asks “Is i > 0?” The answer is yes, so execute the body of the loop.

Repeat the previous printing

and decrementing until you print 1.

. . .

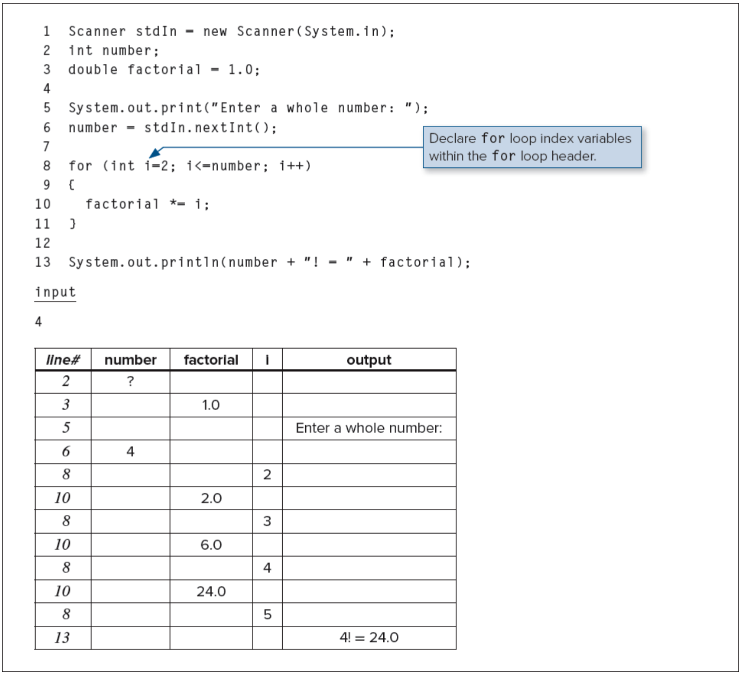
After printing 1, because you’re at the bottom of the loop, decrement i from 1 to 0.

The condition component asks “Is i > 0?” The answer is no, so quit the loop, drop down to the first statement after the closing brace, and print “Liftoff!”

Alternatively, we could have implemented the solution with a while loop or a do loop. Why is the for loop preferable? With a while loop or a do loop, you’d need two extra statements to initialize and update the count variable. That would work OK, but using a for loop is more elegant.

Factorial Example

Now, let's make sure you really understand how the for loop works by studying a formal trace of the second example mentioned at the start of this section—the calculation of a factorial. Figure 4.15 shows the factorial-calculation code listing and its associated trace. Note the input column in the top-left corner of the trace. You didn't have input in Chapter 3's trace examples, so input is worth mentioning now. When the program reads an input value, you copy the next input from the input column into the next row under the variable to which the input is assigned. In this case, when you get to number = stdIn.nextInt(), you copy the 4 from the input column to the next row in the number column.



```
 1  Scanner stdIn = new Scanner(System.in);
 2  int number;
 3  double factorial = 1.0;
 4
 5  System.out.print("Enter a whole number: ");
 6  number = stdIn.nextInt();
 7
 8  for (int i=2; i<=number; i++)
 9  {
10    factorial *= i;
11  }
12
13  System.out.println(number + "! = " + factorial);
```

Declare `for` loop index variables within the `for` loop header.

```
input

4
```

line#	number	factorial	i	output
2	?			
3		1.0		
5				Enter a whole number:
6	4			
8			2	
10		2.0		
8			3	
10		6.0		
8			4	
10		24.0		
8			5	
13				4! = 24.0

Figure 4.15 Code fragment that illustrates factorial calculation plus its associated trace

This trace shows that the 8, 10 sequence repeats three times, so there are indeed three iterations, as expected. Suppose you entered number = 0. Does the program work for that extreme case? The loop header initializes int i=2 and then immediately tests to see if i< =number. Because this condition is false, the loop terminates before it starts, and the code prints the initial value of factorial, which is 1.0. That’s correct, because 0 factorial does indeed equal 1.

Little mistakes are better than big ones.

What about the other extreme case—when the input value is very large? The factorial of a number increases much more rapidly than the number itself increases. If we had declared factorial to be of type int, then input values greater than 12 would cause the factorial variable to overflow, and the output value would be horribly wrong! That’s why we declared factorial to be of type double. A double has more precision than an int, and it gives approximately correct answers even when its precision is inadequate. This makes the program more robust because it fails more *gracefully*. That is, when it fails, it fails just a little bit, not a lot.

Scope of for Loop Index

In the for loop examples presented so far, the loop's index variable (i) is initialized (declared and given an initial value) in the for loop header. This limits the *scope* or recognizable range of the index variable to the for loop itself. In other words, whenever a variable is declared in the for loop header, it exists and can be recognized and used only by code that is within the body of that for loop. For example, if you tried to use the value of the i index variable in the print statement that followed the final brace of the for loop in Figure 4.15, the compiler would say "cannot find symbol. . . variable i."

Sometimes variables used in a loop need to have a scope beyond the loop's scope. The above Factorial program illustrates what we're talking about. The factorial variable must be available for the print statement after the end of the loop, so it must be declared outside the loop. Because it is also needed in the loop, it must be declared before the loop, so we declare it at the beginning of the method with the other variables whose scopes extend throughout the method.

Local Variable Type Inferencing

Regardless of whether you initialize a variable at the top of a method or in a for loop header, you might want to consider an alternative technique, introduced as part of the Java 10 release. Because an initialization includes an assigned value with an identifiable type, the initialization statement doesn't need to restate that type before the variable name. Instead, it may use the generic reserved type name, var, and let the compiler figure out the new variable's actual type with a local variable type inference. For example, in Figure 4.15, instead of:

```
double factorial = 1.0;
```

we could use:

```
var factorial = 1.0;
```

And instead of:

```
for (int i=2; i<=number; i++)
```

we could use:

```
for (var i=2; i<=number; i++)
```

A type name before the variable name in a declaration flags the scope of the variable being declared, and the alternative var flags scope in the same way. This var alternative keeps Java from being outmatched by competitive languages like C++, C#, and Swift, which already use some type inferencing. In principle, this alternative improves programming productivity by making code shorter. This feature is most beneficial when the type is a very long word. However, as our factorial example shows, the code reduction is often marginal. Because replacing a statement’s actual type with var degrades self documentation, we will not use var in subsequent examples. We introduce it here to alert you to its possible use by others.

4.11 Solving the Problem of Which Loop to Use

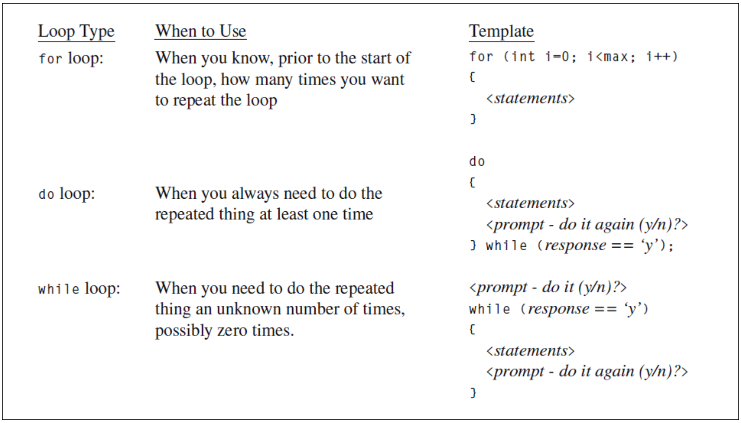
It’s helpful to keep in mind that all three types of loops have an initial condition, a test of that condition, and an update to the tested condition. In while loops and do loops, initialization occurs before the loop starts, and updating occurs within the body of the loop. In for loops, initialization, testing, and updating all occur in the header. The do loop’s decision point is at the bottom of the loop. That's in contrast to the while and for loops, where the decision point is at the top of the loop. When the decision point is at the top of the loop, the decision stands out more and the code is therefore less prone to programming error.

A toolkit needs more than one tool.

With programming, as in life, there are usually many different ways to accomplish the same thing. For example, for a problem that requires repetition, you can actually use any of the three loops to solve any repetition problem. Even though that's the case, you should strive to make your programs elegant, and that means choosing the most appropriate loop even though any loop could be made to work.

Flexibility makes programming fun if you like to be creative. But if you're just starting out, that flexibility can lead to confusion. In Figure 4.16, we provide a table that attempts to alleviate some of that confusion. It suggests a way to choose an appropriate type of loop and how to get started with that loop’s code. We use angled brackets around text to indicate that the enclosed text is a description of code, not actual code. Thus, in using Figure 4.16’s do loop and while loop templates, you’ll need to replace <*prompt—do it again (y/n?)*> with actual code. For example, for a game program, you might use this actual code:

```
System.out.print(”Do you want to play another game (y/n)? ”);
response = stdIn.nextLine().charAt(0);
```



Loop Type | When to Use | Template

`for` loop: When you know, prior to the start of the loop, how many times you want to repeat the loop

```
for (int i=0; i<max; i++)
{
  <statements>
}
```

`do` loop: When you always need to do the repeated thing at least one time

```
do
{
  <statements>
  <prompt - do it again (y/n)?>
} while (response == 'y');
```

`while` loop: When you need to do the repeated thing an unknown number of times, possibly zero times.

```
<prompt - do it (y/n)?>
while (response == 'y')
{
  <statements>
  <prompt - do it again (y/n)?>
}
```

Figure 4.16 Choosing the right loop and getting started with the loop's code

When figuring out which loop to use, it's best to think about the loops in the order of appearance in Figure 4.16. Why? Note how the for loop uses the fewest lines, the do loop uses the next fewest lines, and the while loop uses the most lines. Thus, the for loop is the most compact, and the do loop is the next most compact. But the while loop is more popular than the do loop because its condition is at the beginning of the loop, which makes it more flexible and easier to find. Although you may wish to avoid the do loop because of its relatively awkward structure, in general, you should use the loop that's most appropriate for your particular problem.

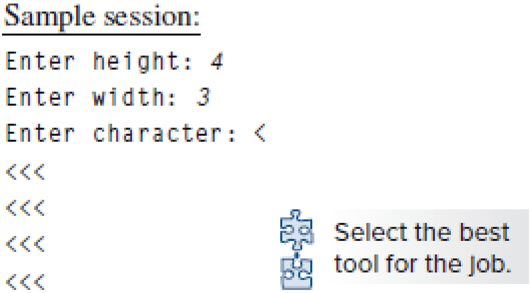
When deciding how to write loop code, you can use the templates shown in Figure 4.16 as starting points. Be aware that in writing loop code, you have to do more than just copy code from Figure 4.16. You need to adapt the code to your particular problem. For example, in writing a for loop, it's common to use i=0 for the initialization component, and that's why the for loop template's initialization component shows i=0. However, if some other initialization component is more appropriate, like count=10, then use the more appropriate code.

4.12 Nested Loops

A *nested loop* is a loop that's inside another loop. You'll see nested loops quite often in real-world programs. In this section, we discuss some of the common characteristics inherent to nested loops.

Suppose you're asked to write a

program that prints a rectangle of characters where the user specifies the rectangle’s height, the rectangle’s width, and the character’s value.



Sample session:

```
Enter height: 4
Enter width: 3
Enter character: <
<<<
<<<
<<<
<<<
```

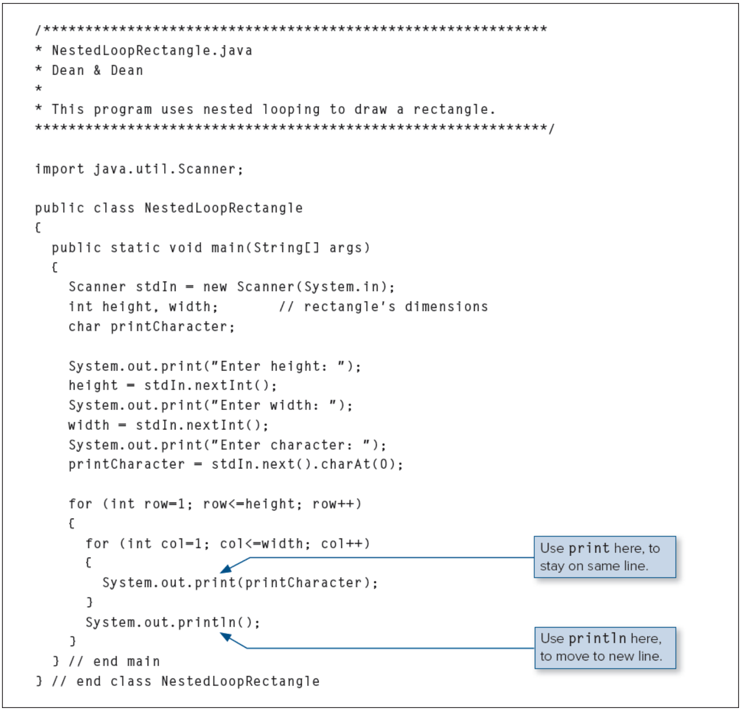
Select the best tool for the job.

To figure out the loops, you first need to think about what needs to be repeated. So, . . . what needs to be repeated? You need to print rows of characters repeatedly. What type of loop should you use to print the rows repeatedly? First, try to use a for loop. The test for a for loop is whether you know the number of times you’ll need to repeat the loop. Do you know the number of times you’ll need to repeat this loop? Yes, the user enters the height, you can use that entered value to determine the number of rows, and that tells you the number of times to repeat the loop. Therefore, you should use a for loop to print successive rows.

Now that you know how to print multiple rows, you need to know how to print an individual row. Do you need to repeat anything when printing an individual row? Yes, you need to print characters repeatedly. So what type of loop should you use for that? Use another for loop because you can use the user's width entry to determine the number of characters to be printed.

So there you go—you need two for loops. Should you put one loop right after the other? No! You need to nest the second loop, the one that prints an individual row, inside the first loop. That should make sense if you word the goal carefully—"Print multiple rows, and within each row, print a sequence of characters." The key word is "within." That tells you to insert the second for loop inside the first for loop's braces.

Using this discussion as a guideline, now write a complete program solution. When you’re done, compare your answer to the NestedLoopRectangle program in Figure 4.17.



```
/*************************************************************
* NestedLoopRectangle.java
* Dean & Dean
*
* This program uses nested looping to draw a rectangle.
*************************************************************/

import java.util.Scanner;

public class NestedLoopRectangle
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int height, width;       // rectangle's dimensions
    char printCharacter;

    System.out.print("Enter height: ");
    height = stdIn.nextInt();
    System.out.print("Enter width: ");
    width = stdIn.nextInt();
    System.out.print("Enter character: ");
    printCharacter = stdIn.next().charAt(0);

    for (int row=1; row<=height; row++)
    {
      for (int col=1; col<=width; col++)
      {
        System.out.print(printCharacter);
      }
      System.out.println();
    }
  } // end main
} // end class NestedLoopRectangle
```

Use `print` here, to stay on same line.

Use `println` here, to move to new line.

Figure 4.17 Program that uses nested loops to draw a rectangle

Note how we use the print method for the print statement inside the inner loop to keep subsequent printed characters on the same line. Then after the inner loop finishes, we use a separate println method to go to the next line.

For most problems where you're dealing with a two-dimensional picture like this rectangle example, you'll want to use nested for loops with index variables named row and col (col is short for column). Why? It makes code more understandable. For example, in the first for loop header, the row variable goes from 1 to 2 to 3, and so on, and that corresponds perfectly with the actual rows printed by the program. However, be aware that it's also common for nested for loops to use index variables named i and j. Why i and j? Because i stands for "index," and j comes after i.

In the NestedLoopRectangle program, there are two levels of nesting, but in general, there may be any number of nesting levels. Each level adds another dimension to the problem. Our NestedLoopRectangle program is quite symmetrical. Both loops are the same type (they're both for loops), and both loops do the same kind of thing (they both help to count the number of times something is printed). In general, however, nested loops do not have to be the same type, and they do not have to do the same kinds of things.

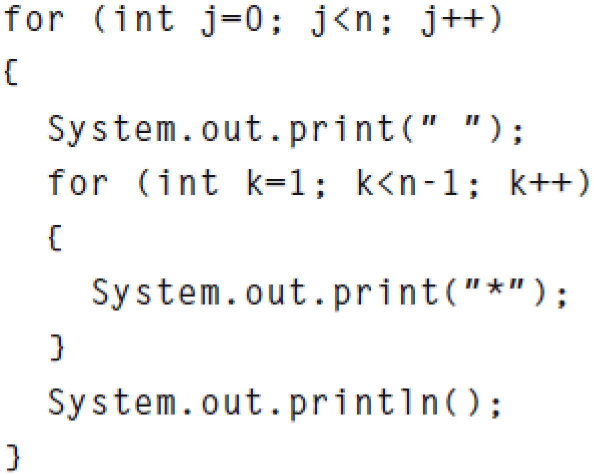
Determining the Number of Iterations in a Nested for Loop

Page 152

As you develop and debug code in the real world, you'll very often need to analyze loops to determine how many times they iterate. If you have a basic for loop that increments its index variable by 1, calculating its iterations is easy: index variable's maximum value minus index variable's minimum value plus 1. For example, in the

NestedLoopRectangle program, the outer loop repeats height − 1 + 1 times or, more simply, height times (where height is row's maximum value, and 1 is row's minimum value). Similarly, you can determine that the NestedLoopRectangle program's inner loop repeats width times. To calculate the number of times the inner loop will execute its print statement, you multiply the outer loop's iterations by the inner loop's iterations. The result indicates that the print statement will be executed height * width times.

Now for a more complex problem. Consider this code fragment:



```
for (int j=0; j<n; j++)
{
  System.out.print(" ");
  for (int k=1; k<n-1; k++)
  {
    System.out.print("*");
  }
  System.out.println();
}
```

To determine how many times the code fragment prints an asterisk, first calculate the outer loop's iterations. Because the outer loop uses $<$ rather than $<=$, the outer loop's maximum index variable value is actually $n - 1$, not n. Thus, the outer loop repeats $(n - 1) - 0 + 1$ times, which simplifies to n. We next calculate the inner loop's iterations: $(n - 2) - 1 + 1$, which simplifies to $n - 2$. Multiplying the outer loop's iterations by the inner loop's iterations yields the formula $n * (n - 2)$.

Think about the boundary cases.

Calculating the number of loop iterations can be tricky, so after you come up with what you think is the final formula, step back and think about the boundary cases. In the example above, $n * (n - 2)$ correctly predicts the number of times the code fragment prints an asterisk, but only when n ≥ 2. What happens when n equals 1? The formula tells you that the code fragment prints −1 asterisks (because 1 * (1 − 2) equals −1), but that is impossible. By examining the inner loop's condition carefully, you should be able to determine that the inner loop iterates 0 times when n equals 1. Likewise, you should be able to determine that the inner loop iterates 0 times when n equals 0. Thus, when n < 2, the code fragment prints 0 asterisks.

4.13 boolean Variables

The conditions that appear in if statements and loops all evaluate to either true or false. We described these Boolean values in Section 4.2. Java also allows us to define a boolean variable, which is a variable that can hold a Boolean value. To declare a boolean variable, specify boolean for the variable's type, like this:

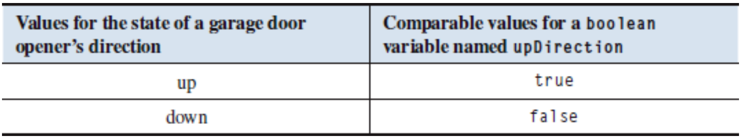
```
boolean upDirection;
```

In this section, we describe when to use boolean variables in general, and we provide a program that uses boolean variables, including the upDirection variable shown above.

When to Use a boolean Variable

Programs often need to keep track of the state of some condition. You can use a boolean variable to keep track of any two-way *state*—a yes/no, up/down, on/off attribute of some entity. For example, if you're writing a program that simulates the operations of an electronic garage door opener, you'll need to keep track of the state of the garage door's direction—is the direction up or down? You need to keep track of the direction "state" because the direction determines what happens when the garage door opener's button is pressed. If the direction state is up, then pressing the garage door button causes the direction to switch to down. If the direction state is down, then pressing the garage door button causes the direction to switch to up.

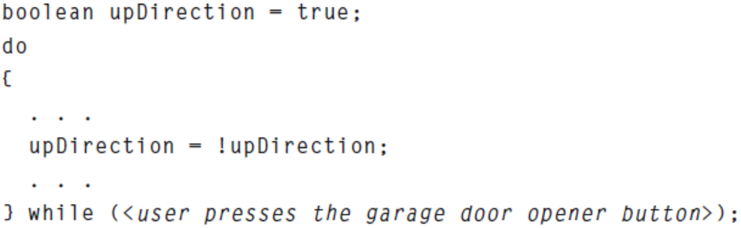
A boolean variable is good at keeping track of the state of some condition when the state has one of two values. For example:



Values for the state of a garage door opener's direction	Comparable values for a `boolean` variable named `upDirection`
up	`true`
down	`false`

Garage Door Opener Example

The following code skeleton illustrates how the upDirection variable work:

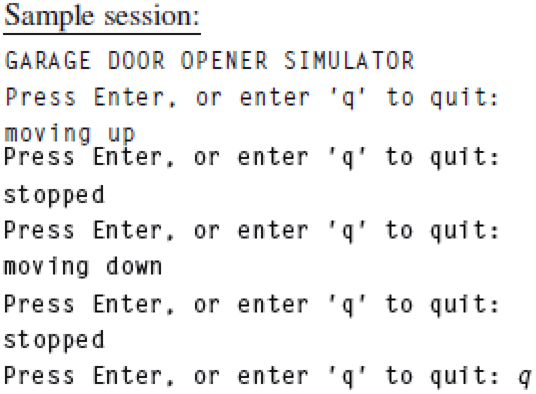


```
boolean upDirection = true;
do
{
  . . .
  upDirection = !upDirection;
  . . .
} while (<user presses the garage door opener button>);
```

The boolean upDirection = true;

statement tells the program to start in the down/closed position and go up when the garage door opener is first pressed. Each iteration of the loop represents what happens when the user presses the garage door opener button. The upDirection = !upDirection statement implements the garage door opener's toggling operation. If upDirection holds the value true, this statement changes it to false, and vice versa.

Now let's look at the upDirection variable in the context of a complete GarageDoor program. In the program, each push of the Enter key on the computer keyboard simulates a push of the garage door opener button. The first push makes the door move upward. The second push makes the door stop. The third push makes the door move downward. The fourth push makes the door stop. And so forth, until the user enters 'q' to make the program quit. Here's an example of user interaction with the GarageDoor program:



Sample session:

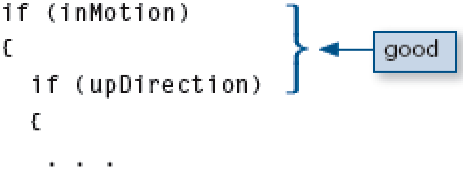
```
GARAGE DOOR OPENER SIMULATOR
Press Enter, or enter 'q' to quit:
moving up
Press Enter, or enter 'q' to quit:
stopped
Press Enter, or enter 'q' to quit:
moving down
Press Enter, or enter 'q' to quit:
stopped
Press Enter, or enter 'q' to quit: q
```

Figure 4.18 contains the program's code. In the program, verify that upDirection is used as previously discussed. Note that there's a second boolean variable, inMotion. The upDirection boolean variable keeps track of the state of going up or down. That one state variable would be good enough if pressing a garage door opener button always generated an up or down motion. But as shown in the sample session, that's not the case. Half the time, pressing the garage door opener causes the garage door to stop moving. Here's the key point: If the door is moving, the door stops, and if the door is stopped, the door starts moving. We keep track of whether the garage door is currently moving with the help of a second state variable, inMotion. The inMotion state variable toggles (goes from false to true or vice versa) at each button push, whereas the upDirection state variable toggles only when the door is stopped—at every other button push.

The GarageDoor program is *user-friendly* because it requires a minimum amount of user input. A given user entry serves one of two purposes. The simplest kind of entry (pressing the Enter key) simulates pushing the button on a garage door opener. Any other entry (not just a 'q' entry) terminates the looping process. Whenever a special data value (in this case anything except a plain Enter) tells a program to stop looping, we say we're using a *sentinel value* to terminate the looping process. Because the program imposes a minimum burden on the user in terms of input, and because the code is relatively concise and efficient, it's appropriate to call this an elegant implementation.

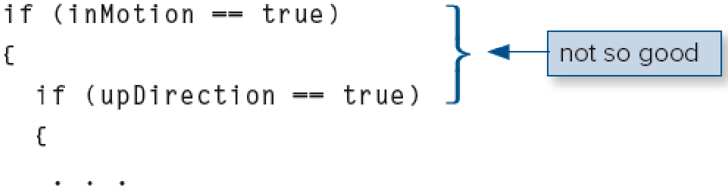
Comparing a boolean Variable Within a Condition

In the GarageDoor program, note how we use the inMotion and upDirection boolean variables by themselves as conditions in if statements:



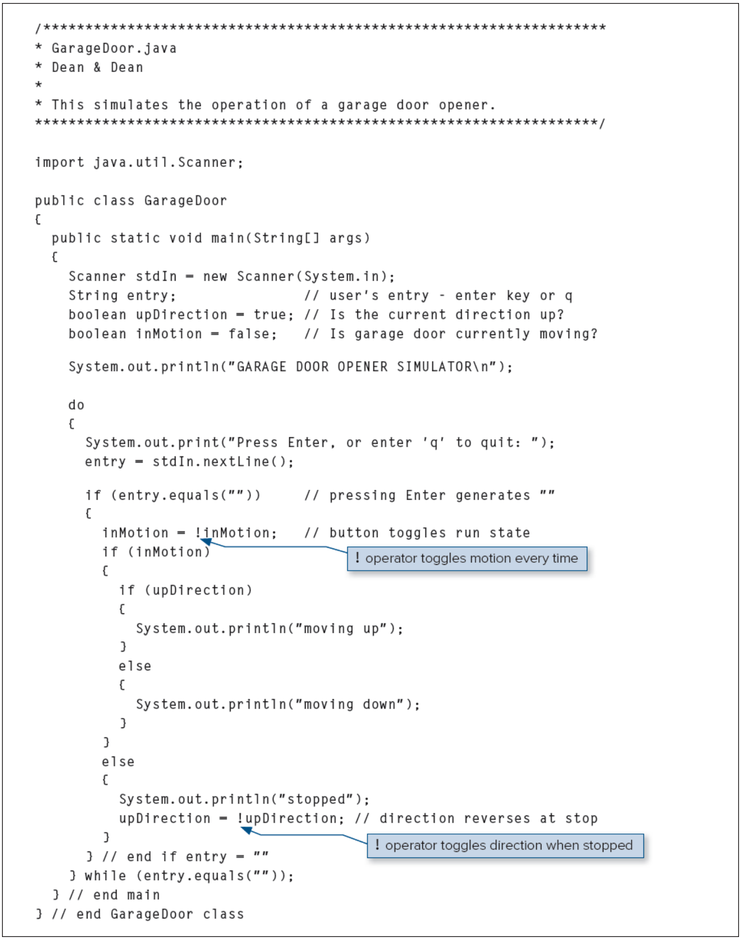
```
if (inMotion)
{
  if (upDirection)
  {
    . . .
```

In the past, you used relational operators in your conditions (<. < =, = =, etc.). But the only rule for a condition is that it needs to evaluate to true or false. A boolean variable is either true or false, so using a boolean variable by itself for a condition is legal. Actually, using a boolean variable by itself for a condition is considered to be elegant. Why? Consider the alternative. The following if conditions are functionally equivalent to the prior if conditions:



```
if (inMotion == true)
{
  if (upDirection == true)
  {
   . . .
```

not so good



```
/*******************************************************************
* GarageDoor.java
* Dean & Dean
*
* This simulates the operation of a garage door opener.
*******************************************************************/

import java.util.Scanner;

public class GarageDoor
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;              // user's entry - enter key or q
    boolean upDirection = true; // Is the current direction up?
    boolean inMotion = false;   // Is garage door currently moving?

    System.out.println("GARAGE DOOR OPENER SIMULATOR\n");

    do
    {
      System.out.print("Press Enter, or enter 'q' to quit: ");
      entry = stdIn.nextLine();

      if (entry.equals(""))     // pressing Enter generates ""
      {
        inMotion = !inMotion;   // button toggles run state
        if (inMotion)
        {
          if (upDirection)
          {
            System.out.println("moving up");
          }
          else
          {
            System.out.println("moving down");
          }
        }
        else
        {
          System.out.println("stopped");
          upDirection = !upDirection; // direction reverses at stop
        }
      } // end if entry = ""
    } while (entry.equals(""));
  } // end main
} // end GarageDoor class
```

! operator toggles motion every time

! operator toggles direction when stopped

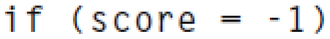
Figure 4.18 The GarageDoor program

Page 156

What would happen if instead of using the double equals (= =), you accidentally used the single equals (=)? The expressions in the two conditions would assign true to inMotion and upDirection, respectively. Then both conditions would evaluate to true (because an assignment operation evaluates to the assigned value), and code in the if statements would execute. That would give the correct result only when inMotion and upDirection were both true beforehand. But it would give an incorrect result if either inMotion or upDirection was false beforehand. Moreover, the unintended alteration of the values of inMotion and upDirection might upset other conditional evaluations. This kind of logic error is insidious because the compiler would not see it, and it would be hard for you to see it too, because = looks so much like ==.

Thus, to steer clear of accidentally using the assignment operator, we recommend that you never compare boolean variables to true or false. Instead, if you need to use a boolean variable within a condition, just use the boolean variable by itself.

By the way, for non-Boolean variables (like int variables and double variables), accidentally using the assignment operator within a condition isn't as much of a problem. What happens if you enter the following code?



```
if (score = -1)
```

Because the code assigns the integer −1 into score, the condition evaluates to an integer. The compiler requires conditions to evaluate to Boolean values, so the code generates a compilation error. Such a compilation error should be relatively easy to identify and fix. Certainly much easier to deal with than the nasty logic error due to assigning true or false to a boolean variable within a condition.

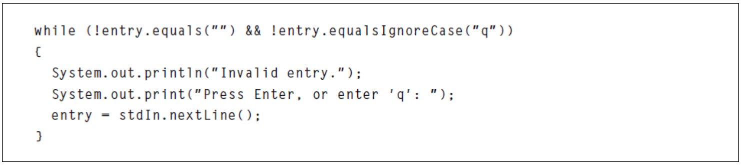
4.14 Input Validation

In the previous section, you learned to use a boolean variable to keep track of a two-way state. In this section, you'll learn to use a boolean variable for a particularly common type of two-way state—the state of a user's input in terms of whether it's valid or invalid.

Input validation is when a program checks a user's input to make sure it's valid (that is, correct and reasonable). If it's valid, the program continues. If it's invalid, the program enters a loop that warns the user about the erroneous input and then prompts the user to re-enter.

⚠ In the GarageDoor program, note how the program checks for an empty string (which indicates the user wants to continue). If the string is not empty, it assumes that the user entered a 'q', but it doesn't check specifically for a 'q'. Consequently, it does not deal well with the possibility that the user accidentally hits another key before pressing the Enter key. It interprets that input as a quit command instead of a mistake.

To make the program more robust, you should provide input validation. There are several possible ways to do this. One of the simplest ways is to insert a while loop whose condition and's together all bad possibilities and whose body warns the user about the erroneous input and then prompts the user to re-enter. For the GarageDoor program in Figure 4.18, input validation is provided by the code fragment in Figure 4.19.



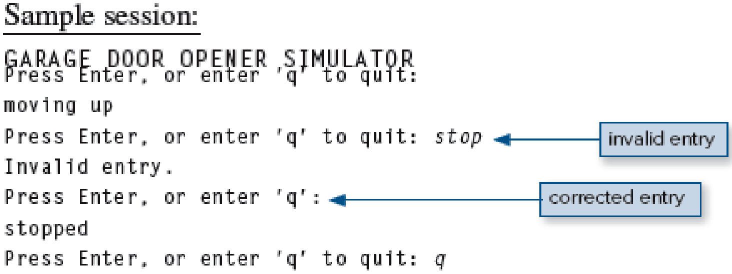
```
while (!entry.equals("") && !entry.equalsIgnoreCase("q"))
{
  System.out.println("Invalid entry.");
  System.out.print("Press Enter, or enter 'q': ");
  entry = stdIn.nextLine();
}
```

Figure 4.19 Input validation loop to insert after the input statement in Figure 4.18

Where should you insert this code fragment? You want to validate the input right after the input is entered. So to make the GarageDoor program more robust, you should insert the above code fragment into Figure 4.18 immediately after this statement:

```
entry = stdIn.nextLine();
```

Running the modified program produces the following sample session:



Sample session:

```
GARAGE DOOR OPENER SIMULATOR
Press Enter, or enter 'q' to quit:
moving up
Press Enter, or enter 'q' to quit: stop
Invalid entry.
Press Enter, or enter 'q':
stopped
Press Enter, or enter 'q' to quit: q
```

Optional Forward References

At this point, some readers might want to learn about *arrays*. An array is a collection of related items of the same type. Array manipulations require the use of loops. As such, arrays provide a means for readers to gain further practice with the material presented in this chapter, specifically the loop material. You’re not required to learn about arrays just yet, but if you can’t wait, you can read about arrays in Chapter 9, Sections 9.1 through 9.6.

Later in the book, we present relatively advanced syntax details that pertain to control statements. For example, embedding an assignment expression in a loop header or using a break statement to break out of a loop. You’re not required to learn those details just yet, but if you can’t wait, you can read about them in Chapter 12, Sections 12.6 through 12.12.

4.15 Problem Solving with Boolean Logic (Optional)

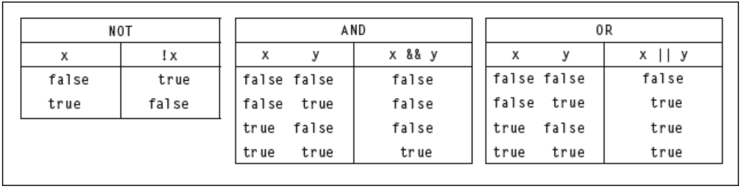
Make the logic as clean as possible.

The conditions for if statements and loops can sometimes get complicated. For a better understanding of complicated conditions, we’ll now look at the logic that comprises a condition. Learning how to manipulate logic should help you to (1) simplify condition code and (2) debug logical problems. You’ve seen how logical operators work when applied to comparison-operator conditions. For example, this code (which uses the operator in conjunction with the > = and < = comparison operators) probably already makes sense to you:

```
(temp >= 50.0 && temp <= 90.0)
```

Now you’ll see how logical operators work with boolean variables. This is called *Boolean logic* or *Boolean algebra*. The most primitive and most general way to describe Boolean logic is with a *truth table*. A truth table is a list of all input combinations and their corresponding outputs. Figure 4.20 presents truth tables for the three basic building blocks of Boolean logic, the logical operators ! for NOT, for AND,

and || for OR.



NOT	
x	!x
false	true
true	false

AND		
x	y	x && y
false	false	false
false	true	false
true	false	false
true	true	true

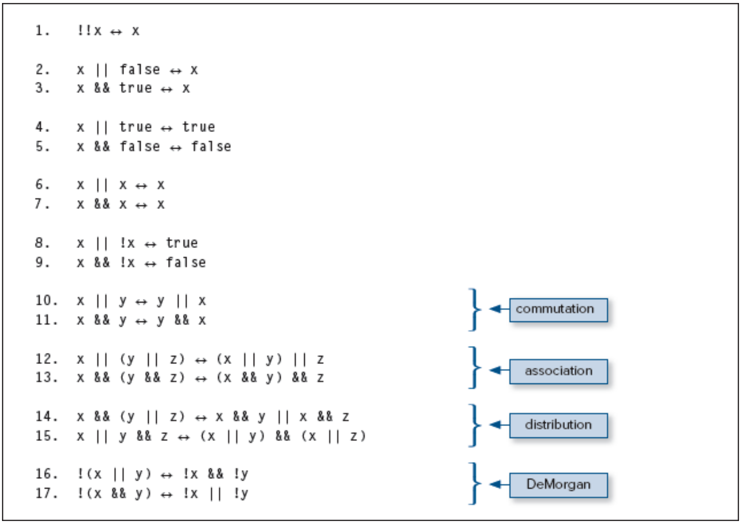
OR		
x	y	x || y
false	false	false
false	true	true
true	false	true
true	true	true

Figure 4.20 Truth tables for the three basic logical operations

Page 158

Boolean Algebra Basic Identities

Sometimes, however, a logical expression is harder to understand. This is particularly true when it includes several “not” (!) operators. To gain a better understanding of what the code means and is supposed to do, it’s sometimes helpful to transform the logical expression to another form. Boolean algebra provides a special set of formulas called *basic identities,* which anyone can use to make transformations. These basic identities are listed in Figure 4.21. The precedence of the various operators is the precedence given in Figure 4.6. That is, ! has the highest precedence, has the next highest precedence, and || has the lowest precedence. The ↔ symbol means equivalence; that is, whatever is on the left side of the double arrow can be replaced by whatever is on the right side, and vice versa.



```
1.   !!x ↔ x

2.   x || false ↔ x
3.   x && true ↔ x

4.   x || true ↔ true
5.   x && false ↔ false

6.   x || x ↔ x
7.   x && x ↔ x

8.   x || !x ↔ true
9.   x && !x ↔ false

10.  x || y ↔ y || x
11.  x && y ↔ y && x

12.  x || (y || z) ↔ (x || y) || z
13.  x && (y && z) ↔ (x && y) && z

14.  x && (y || z) ↔ x && y || x && z
15.  x || y && z ↔ (x || y) && (x || z)

16.  !(x || y) ↔ !x && !y
17.  !(x && y) ↔ !x || !y
```

commutation
association
distribution
DeMorgan

Figure 4.21 Basic identities of Boolean algebra You can use these identities in any combination to change the form of any conditional expression.

Page 159

The first 13 identities are relatively

straightforward, and you should be able to satisfy yourself of their validity by just thinking about them. Likewise, you shouldn't have to memorize them. You should be able to use them instinctively. For example, *commutation* means you can switch the order without changing anything, and *association* means you can move the parentheses without changing anything. The last four identities are more mysterious, and some of them might even seem unreasonable at first. For example, *distribution* is a kind of shuffling, and *DeMorgan's* laws say you can negate everything and exchange all and's and or's.

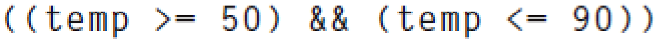
Proving the Boolean Identities

Now that you've seen the basic identities, let's see how to prove them. The proof technique is to write a program that compares two arbitrary logical expressions for all possible values of the boolean variables they contain. If the two expressions evaluate to the same truth values for all possible variable values, they are logically equivalent. Figure 4.22 contains a program that does just that for the special case of the expressions on either side of basic identity 16 in Figure 4.21.

It's straightforward to modify the TruthTable program in Figure 4.22 to test any of the other basic identities in Figure 4.21. In fact, you can modify the program to test any prospective logical equivalence. To test a different equivalence, substitute the left and right sides of the prospective equivalence for the expressions assigned to result1 and result2, respectively.

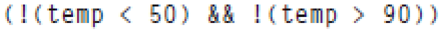
Applications

There are many ways you can use Boolean identities. For example, consider the condition in the if statement in Figure 4.5, which looked like this:



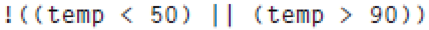
```
((temp >= 50) && (temp <= 90))
```

If you apply the ! operator to a comparison operator expression, that causes the expression’s comparison operator to change. For example, !(temp > = 50) is equivalent to (temp < 50). With that reasoning in mind plus basic identity 1, you can apply !! to each of the above comparison expressions and generate this equivalent condition:



```
(!(temp < 50) && !(temp > 90))
```

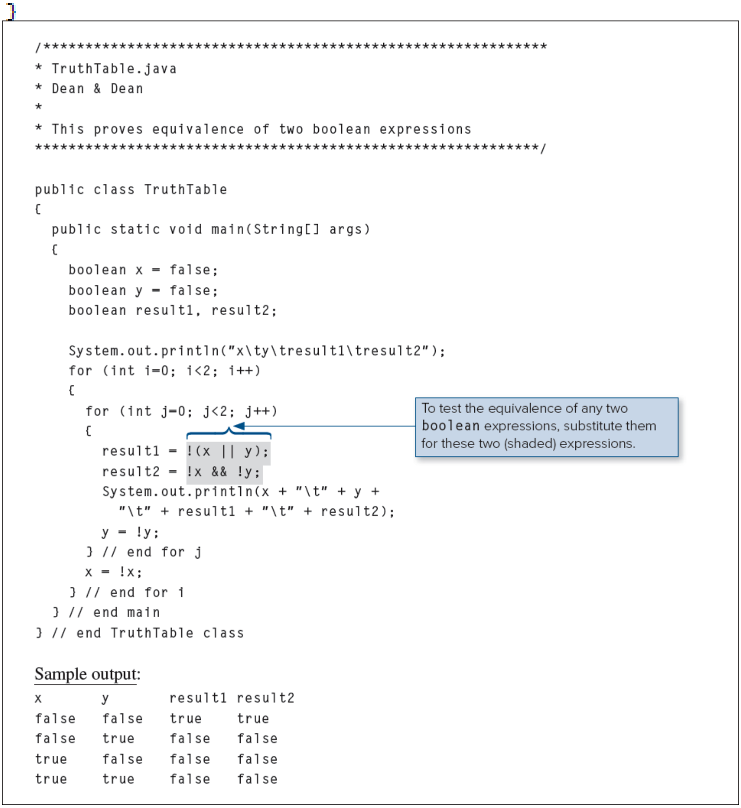
You can apply basic identity 16 to the above condition and come up with this equivalent condition:



```
!((temp < 50) || (temp > 90))
```

You can use the above condition as part of a replacement for Figure 4.5’s original if statement where the if and else subordinate statements are swapped. Here's the resulting functionally equivalent if statement:

```
if ((temp < 50) || (temp > 90))
{
  System.out.println("not OK");
}
else
{
  System.out.println("OK");
}
```



```
/*************************************************************
* TruthTable.java
* Dean & Dean
*
* This proves equivalence of two boolean expressions
*************************************************************/

public class TruthTable
{
  public static void main(String[] args)
  {
    boolean x = false;
    boolean y = false;
    boolean result1, result2;

    System.out.println("x\ty\tresult1\tresult2");
    for (int i=0; i<2; i++)
    {
      for (int j=0; j<2; j++)
      {
        result1 = !(x || y);
        result2 = !x && !y;
        System.out.println(x + "\t" + y +
          "\t" + result1 + "\t" + result2);
        y = !y;
      } // end for j
      x = !x;
    } // end for i
  } // end main
} // end TruthTable class
```

To test the equivalence of any two `boolean` expressions, substitute them for these two (shaded) expressions.

Sample output:

```
x       y       result1 result2
false   false   true    true
false   true    false   false
true    false   false   false
true    true    false   false
```

Figure 4.22 Program that generates a truth table for two logical expressions If result1 and result2 values are the same in all rows, the expressions are

equivalent.

For another example, consider the condition in the while loop in Figure 4.19, which looks like this:

```
(!entry.equals("") && !entry.equalsIgnoreCase("q"))
```

You can apply basic identity 16 to the above condition and come up with this equivalent condition:

```
!(entry.equals("") || entry.equalsIgnoreCase("q"))
```

Chapter Summary

- You can alter a program’s sequence of execution by using an if statement. The choice of which of two alternative paths to take is determined by the truth of the if statement’s condition.
- Use the “if, else if” form of the if statement to choose among three or more alternatives.

- You must use braces around two or more subordinate statements within any part of an if statement, and it’s advisable to use them even when there is only one subordinate statement.

- A condition’s comparison operators (<, >, <=, >=, ==, and !=) have higher priority than its “and” (&&) and “or” (||) logical operators.

- To negate the result of and/or || operations, enclose them in parentheses and precede them with a ! operator.

- Use a switch statement or a switch expression to choose among several alternatives and make that choice by attempting to match an int, a char, or a String.

- If the condition in a while loop’s header is true, whatever is in the subsequent block executes, and then if the condition is still true, that execution repeats.

- A do loop executes its block at least once, and it repeats that execution as long as the condition after the final while remains true.
- A for loop executes its block as long as the condition in the second component of its header remains true. The first component in the header initializes a count variable before the first execution, and the third component in the header updates that count variable after each execution and before the next evaluation of the second component's condition.
- You can perform multidimensional iteration by putting loops inside other loops.
- To avoid duplication and/or clutter, assign complicated logical expressions to boolean variables,

and use those variables in if statement or looping conditions.

- Use input validation to avoid bringing bad data into your programs.
- Optionally, use Boolean logic to simplify the expressions in if statement and looping conditions, and use truth tables to verify the equivalence of alternative logical expressions.

Review Questions

§4.2 Conditions and boolean Values

1. What are Java’s two Boolean values?

2. Provide a list of Java’s comparison operators.

§4.3 if Statements

3. Provide an if statement that implements this logic:

When the water temperature is less than 120 °F, turn the heater on by assigning the value “on” to a heater string variable. When the water temperature is greater than 140 °F, turn the heater off by assigning the value “off” to a heater string variable. Don’t do anything when the water temperature is between these two temperatures.

4. What is the maximum number of “else if” blocks allowed in an if statement that uses the “if, else if” form?

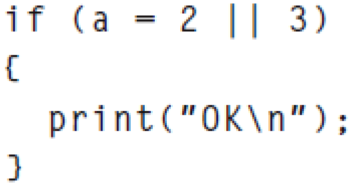
§4.4 && Logical Operator

5. The relational and equality operators have higher precedence than the arithmetic operators. (T / F)

§4.5 || Logical Operator

Page 162

6. Correct the following code fragment so that it executes and outputs OK if a, an int variable, is equal to either 2 or 3:



```
if (a = 2 || 3)
{
  print("OK\n");
}
```

§4.6 ! Logical Operator

7. What Java operator reverses the truth or falsity of a condition?

§4.7 Switching Constructs

8. Suppose a switch statement's controlling expression is (stdIn.next().charAt(0)), and you want a user input of 'Q' or 'q' to execute the same statement, which is: System.out.println("quitting"); Write the code fragment for a case

clause that does that.

9. Write the header for a switch expression in which each case clause responds to an int variable named dayOfWeek and returns a value that's assigned to a variable named avgTemp.

10. Suppose you have an 'if, else" form of the if statement. If you want to replace it with a switch construct, can you use the if statement's condition as the controlling expression in the switch header?

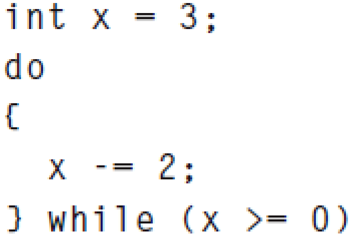
§4.8 while Loop

11. What must a while loop condition evaluate to?

12. Suppose you want to use the user-query technique to terminate a simple while loop. Where should you put the user query?

§4.9 do Loop

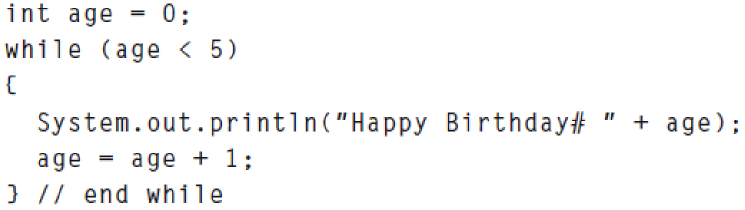
13. What's wrong with this code fragment?



```
int x = 3;
do
{
  x -= 2;
} while (x >= 0)
```

14. If you know ahead of time the exact number of iterations through a loop, what type of loop should you use?

15. Implement the following as a for loop:



```
int age = 0;
while (age < 5)
{
  System.out.println("Happy Birthday# " + age);
  age = age + 1;
} // end while
```

§4.11 Solving the Problem of Which Loop to Use

16. If you know that a loop should be executed at least one time, what type of loop is most appropriate?

§4.12 Nested Loops

Page 163

17. Construct a template for a for loop inside a for loop. Use i for the outer for loop's index variable and use j for the inner for loop's index variable.

§4.13 Boolean Variables

18. Assume that the variable OK has been declared to be of type boolean. Replace the following code with an equivalent for loop:

```
OK = false;
while (!OK)
{
  <statement(s)>
}
```

§4.15 Problem Solving with Boolean Logic (Optional)

19. Given the logical expression:

```
!(!a || !b)
```

Replace it with an equivalent

logical expression that is completely devoid of "not" operations.

Exercises

---

1. [after §4.8] Whenever you cook eggs for breakfast, you must decide how many eggs to use. You like to use this rule of thumb—use approximately one and a half eggs per person. Of course, you can't use half an egg, so, you round up. For example, if you have three people who want to eat, you use five eggs.

Write a code fragment that prompts the user to enter the number of eaters, calculates the number of eggs required, and prints either "No one wants to eat." (if the user enters a nonpositive value) or "Go to the Waffle House!" (if the user enters a value greater than 3) or “Use number-of-eggs eggs.”, where number-of-eggs is the number of eggs needed to feed the specified number of eaters.

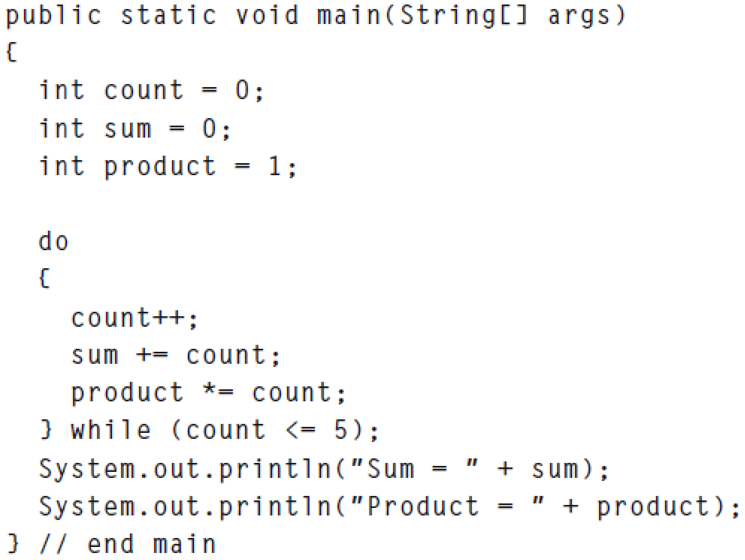
2. [after §4.3] Given this code fragment:

```
1    double x = -0.5;
2
3    while (x * x <= 40)
4    {
5      switch ((int) x)
6      {
7        case 4, 3, 2 -> System.out.println(”x= ” + x);
8        default ->
9          System.out.println(”something else, x= ” + x);
10     } // end switch
11     x += 2;
12   } // end while
```

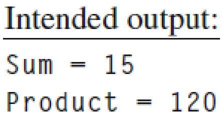
Trace the code using either the short form or the long form. To help you get started, here’s the trace setup. The third column is for switch’s controlling expression. For the short form, you won’t need the line# column.

line#	x	(int) x	output

3. [after §4.9] The following main method is supposed to print the sum of the numbers 1 through 5 and the product of the numbers 1 through 5. Identify the bug. Although not required, to test your answer, you are encouraged to enter your corrected code in a program and run it.



```
public static void main(String[] args)
{
  int count = 0;
  int sum = 0;
  int product = 1;

  do
  {
    count++;
    sum += count;
    product *= count;
  } while (count <= 5);
  System.out.println(”Sum = ” + sum);
  System.out.println(”Product = ” + product);
} // end main
```



Intended output:

```
Sum = 15
Product = 120
```

4. [after §4.10] Given this main method:

```
1    public static void main(String[] args)
2    {
3      int i;
4      String debug = "";
5      for (i=3; i>0; i--)
6      {
7        debug = switch (i + 1)
8        {
9          case 1 -> "one";
10         case 2, 3 -> "two or three";
11         case 4  -> "four";
12         default -> "default";
13       }; // end switch
14       System.out.println(debug);
15     } // end for
16   } // end main
```

Trace the code using the long form. To help you get started, here's the trace setup.

line#	i	i+1	debug	output

The third column is for the switch expression's controlling expression. When evaluating a switch expression, under line# provide both the line where the expression is evaluated and the line where that value is assigned, like this:

10, 7			two or three	

5. [after §4.10] Given the below program skeleton, insert code where it says <insert code here> such that the program prints the sum of odd integers from 1 to num. You are not required to perform input validation.

```
public class SumOddInts
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int num, sum;

    System.out.print("Enter a positive odd number: ");
    num = stdIn.nextInt();

    <insert code here>

    System.out.println("Sum = " + sum);
  } // end main
} // end class SumOddInts
```

Sample session:

```
Enter a positive odd number: 9
Sum = 25
```

6. [after §4.12] Given this main method:

```
1    public static void main(String[] args)
2    {
3      for (int start=8; start>1; start-=4)
4      {
5        for (int count=start; count>0; count--)
6        {
7          System.out.println(count);
8        }
9        System.out.println("Liftoff!");
10     }
11   } // end main
```

Trace the code using either the short form or the long form. To help you get started, here's the trace setup. For the short form, you won't need the line# column.

line#	start	count	output

7. [after §4.12] Given the following code fragment, and assuming that n is an int, write an expression for the total number of lines of output as a function of n.

```
for (int j=0; j<=n; j++)
{
  for (int k=n; k>0; k--)
  {
    System.out.println("***");
  }
}
```

What is the smallest value of n that produces some output, and how many lines does this produce?

8. [after §4.13] Given this main method:

```
1   public static void main(String[] args)
2   {
3     boolean loves = false;
4
5     for (int num=0; num<4; num++)
6     {
7       if (loves)
8       {
9         System.out.println(”She loves me!”);
10      }
11      else
12      {
13        System.out.println(”She loves me not!”);
14      }
15      loves = !loves;
16    }
17  } // end main
```

Trace the code using either the short form or the long form. To help you get started, here’s the trace setup. For the short form, you won’t need the line# column.

line#	loves	num	output

9. [after §4.13] Consider the TestScores program below.

```
/*********************************************************
* TestScores.java
* Dean & Dean
*
* This implements an algorithm that averages test scores.
*********************************************************/

import java.util.Scanner;

public class TestScores
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int score;
    int scoreSum = 0;
    int count = 0;
    double average = 0;
    System.out.print("Enter score (-1 to quit): ");
    score = stdIn.nextInt();

    while (score >= 0)
    {
      scoreSum += score;
      count++;
      average = (double) scoreSum / count;
      System.out.print("Enter score (-1 to quit): ");
      score = stdIn.nextInt();
    }
    System.out.println("Average score is " + average);
  } // end main
} // end TestScores class
```

In the above program, note how the following lines appear above the loop and also at the bottom of the loop:

```
System.out.print("Enter score (-1 to quit): ");
score = stdIn.nextInt();
```

Modify the program so it avoids having those lines appear twice but keeps the same functionality. For your modification, you are required to use a boolean variable named done in the while loop's condition.

10. [after §4.13] Consider the following code fragment. Without changing the loop type, modify the code to prevent output when the input equals the sentinel value of zero.

```
Scanner stdIn = new Scanner(System.in);
int x;

do
{
  x = stdIn.nextInt();
  System.out.println("negative = " + (-x));
} while (x != 0);
```

11. [after §4.15] Here's a brain teaser that uses Boolean logic:

You’re traveling on a road, and you come to a fork in the road. You know that one path leads to a pot of gold and the other path leads to a dragon. There are two elves at the fork, both of whom know the way to the pot of gold. You know that one elf always tells the truth and the other elf always lies, but you don’t know which elf is which. What single question should you ask to figure out the proper path to the pot of gold?

Review Question Solutions

---

1. Java’s Boolean values are true and false.

2. Java’s comparison operators are:

`==, !=, <, >, <=, >=`

3. Use an “if, else if” statement,

like this:

```
if (temp < 120)
{
  heater = "on";
}
else if (temp > 140)
{
  heater = "off";
}
```

Page 168

Do not include a final else.

4. There is no limit on the number of “else if” blocks that are allowed.

5. False. The arithmetic operators have higher precedence than the comparison operators.

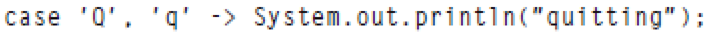
6. The corrections are underlined:

```
(a == 2 || a == 3)
{
  System.out.print("OK\n");
}
```

7. The ! operator reverses the truth or falsity of a condition.

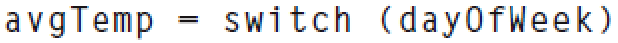
8. If a switch statement’s

controlling expression is (stdIn.next().charAt(0)), and you want ‘Q’ or ‘q’ to execute the print statement, use:



```
case 'Q', 'q' -> System.out.println("quitting");
```

9. Header for a switch expression in which each case clause returns a value that’s assigned to an avgTemp variable:



```
avgTemp = switch (dayOfWeek)
```

10. No. An if statement’s condition evaluates to either true or false. The controlling expression in a switch header must evaluate to one of the allowed types (e.g., int, char, String), and boolean is not one of the allowed types.

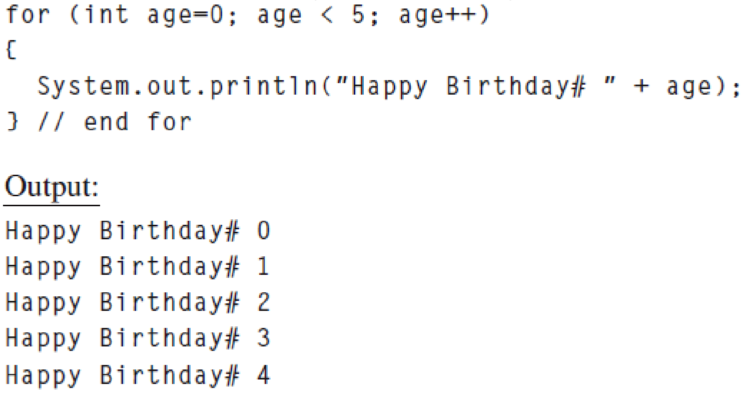
11. A while condition evaluates to either true or false.

12. The user query should occur just prior to where the termination condition is tested. A while loop tests the termination condition at the beginning of the loop. Therefore, the user query should occur just above the top of the loop and also just above the bottom of the loop. If you want the loop to always execute at least once, then omit the user query above the loop and replace it with an assignment that forces the termination condition to be true.

13. There is no semicolon after the while condition.

14. If you know ahead of time the exact number of iterations through a loop, use a for loop.

15. Happy birthday as a for loop:



```
for (int age=0; age < 5; age++)
{
  System.out.println("Happy Birthday# " + age);
} // end for
```

Output:

```
Happy Birthday# 0
Happy Birthday# 1
Happy Birthday# 2
Happy Birthday# 3
Happy Birthday# 4
```

16. A do loop is most appropriate in simple situations where there will always be at least one pass.

17. Template for a pair of nested for loops:

```
for (int i=0; i<imax; i++)
{
  for (int j=0; j<jmax; j++)
  {
    <statement(s)>
  } // end for j
} // end for i
```

18. A for loop representation of a while loop:

```
for (boolean OK=false; !OK;)
{
  <statement(s)>
}
```

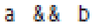
19. Given the expression:

```
!(!a || !b)
```

Starting on the left side of basic identity 16 and going to the right side gives this:

```
!!a && !!b
```

Then using basic identity 1 gives this:



```
a && b
```

---

[1] The iPod is a portable media player designed and marketed by Apple Computer.

[2] We describe enumerated types in Chapter 12. Other permitted types are the primitive types byte and short, and the wrapped types Byte, Short, Integer, and Character. We describe byte and short in Chapter 12, and we describe wrapped types in Chapter 5.

[3] https://www.unitedstateszipcodes.org.

Using Prebuilt Methods

Objectives

- See what it takes to incorporate Java’s prebuilt application programming interface (API) software into your programs, and become acquainted with Oracle’s documentation of the API software.

- Use the methods and named constants defined in Java’s Math class.
- Use the parsing methods in wrapper classes to convert text representations of numbers into - numerical format, and learn to use the toString methods to go the other way.
- Use methods in the Character class to identify and alter character types and formats.
- Use methods in the String class to find the first index of a particular character, extract or replace substrings, convert case, and trim leading and trailing whitespaces.
- Format output with the System.out.printf method.
- Optionally use the Random class to generate nonuniform random-number distributions.

- Optionally display an image covered by a semitransparent pane with nonuniform opacity.

Outline

5.1 Introduction 170

5.2 The API Library 171

5.3 Math Class 177

5.4 Wrapper Classes for Primitive Types 182

5.5 Character Class 186

5.6 String Methods 188

5.7 Formatted Output with the printf Method 194

5.8 Problem Solving with Random Numbers (Optional) 199

5.9 GUI Track: Covering an Image with a Tinted Pane

(Optional) 203

5.1 Introduction

In Chapters 3 and 4, we focused on basic Java programming language constructs—variables, assignments, operators, if statements, loops, and so on. We also introduced a more advanced programming technique—calling a method. Method calls provide a lot of "bang for your buck." In other words, they do a lot and require very little work on your part. For example, you get great benefit for little effort when you call the print and println methods for output, the next, nextLine, nextInt, and nextDouble methods for input, and the charAt, length, equals, and equalsIgnoreCase methods for string manipulation. In this chapter, we want to expose you to other methods that are already written, already tested, and are readily accessible to all Java programmers.

While this chapter raises your awareness of valuable already written methods, it also gives you a better feeling for what methods can do in general. And learning what methods can do is an important first step in learning about *object-oriented programming* (OOP). We describe OOP in all its glory in the next chapter, but for now, here's a pared-down explanation: OOP is the idea that programs should be broken down into objects. An *object* is a set of related data plus a set of behaviors. For example, a string is an object: A string's "set of related data" is its characters, and its "set of behaviors" is its methods (the length method, the charAt method, etc.). Each object is an instance of a class. For example, a single string object, "hello," is an instance of the String class. This chapter serves as a transition from Java basics in Chapters 3 and 4 to OOP in the remainder of the book. We carry out this transition by showing you how to use prebuilt OOP code without having to implement it yourself. More specifically, in this chapter, you learn how to use methods, and in the next chapter, you'll learn how to write your own classes and the methods that go inside those classes.

There are two basic types of methods, *instance methods* and *static methods,* and we provide examples of both in this chapter. Instance methods are methods that are associated with a particular instance of a class. For example, to call the String class's length method, you have to associate it with a particular string. So in the example below, note how the firstName string is associated with the length method:

```
firstNameSize = firstName.length();
```

The firstName string is an example of a *calling object*. As the name implies, a calling object is an object that calls a method. Whenever you call an instance method, you have to prefix the method name with a calling object and then a dot.

Static methods are methods that are associated with an entire class, not with a particular instance of a class. For example, there's a Math class that contains many static methods. Its methods are associated with math in general, not with a particular instance of math (referring to a particular instance of math doesn't even make sense). To call a static method, you prefix the method name with the name of the class that defines it. For example, the Math class contains a round method that returns the rounded version of a given value. To call the round method, you prefix it with Math like this:

```
paymentInDollars = Math.round(calculatedEarnings);
```

We start the chapter with an overview of the API library, which is Oracle’s collection of prebuilt classes. We then examine the Math class, which provides methods for mathematical calculations. We next turn our attention to the wrapper classes, which encapsulate (wrap up) primitive data types. We then expand on our previous discussion of the String class by providing additional string methods. After that, we describe the printf method, which provides formatted output functionality. We then discuss the Random class, which provides methods for generating random numbers. We end the chapter with an optional GUI track section. In it, we discuss methods provided by API classes in image, shape, and paint packages and describe how to manipulate and display images and graphics. Very cool stuff!

5.2 The API Library

When working on a programming problem, you should normally check to see if there are prebuilt classes that meet your program's needs. If there are such prebuilt classes, then use those classes—don't "reinvent the wheel." For example, user input is a rather complicated task. Java's Scanner class handles user input. Whenever you need user input in a program, use the Scanner class rather than writing and using your own input class.

There are two primary advantages of using prebuilt classes. Using prebuilt classes can save you time because you don't have to write the classes yourself. Using prebuilt classes can also improve the quality of your programs because the classes have been thoroughly tested, debugged, and scrutinized for efficiency, and other programmers will find it easier to understand the programs you write.

Searching API Class Library Documentation

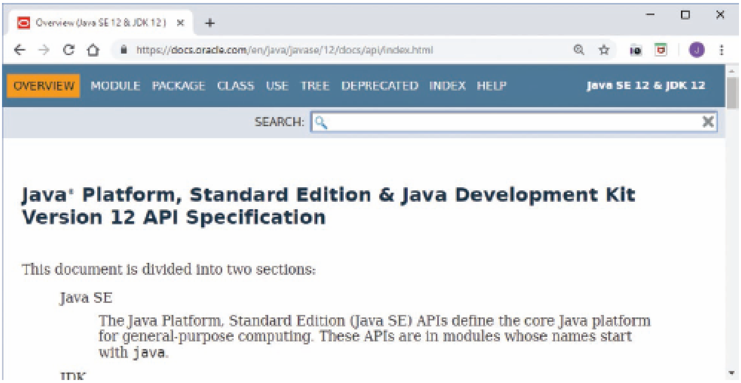
Java's prebuilt classes are stored in the Application Programming Interface (API) library, which is more simply known as the Java API library. You should be able to find documentation for the API library at Oracle's Java API website:

https://docs.oracle.com/en/java/javase/12/docs/api/index.html

At the time of this edition's writing, Java's release number was 12, which explains the 12 in the above website address. With Oracle's plan for new releases every six months, by the time you read this, there will be Java API website addresses with larger numbers than 12 in them. Things don't change all that much with each new Java release, so what you learn in this edition should coincide with what you see in the later Java API websites.

The API library contains thousands of classes and those classes contain tens of thousands of methods. It’s unlikely that you’ll be able to memorize the names, syntax, and functionality of all those classes and methods, so how do you locate the particular piece of prebuilt software that might be just what you need for your current programming project? Use a textbook (like this one ☺) to get you started with selected sample classes and methods. Then go to the Java API website and browse.

Figure 5.1a shows the website’s initial display. Note the search box in the top-right corner. If you’d like to learn about a particular class or method and you think you might know its name, type the name in the search box, and as you type, you’ll see matches pop up. For example, if you type “next,” all the classes, named constants, and methods that have “next” in their names will pop up (e.g., the nextInt and nextLine methods will pop up). Figure 5.1b shows what happens if you type “nextint” (the search is case-insensitive, so “nextint” finds the nextInt method). Note that there are several nextInt methods. The one that we’ve been using in prior programs is the one labeled java.util.Scanner.nextInt() in the figure.


Overview (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/index.html
OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Java SE 12 & JDK 12
SEARCH:

Java® Platform, Standard Edition & Java Development Kit Version 12 API Specification

This document is divided into two sections:

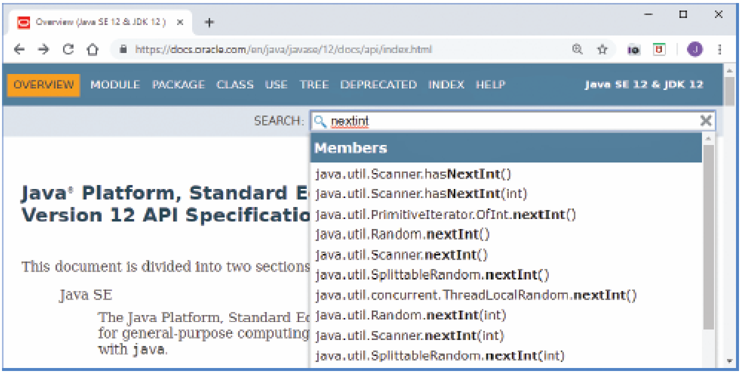
Java SE

The Java Platform, Standard Edition (Java SE) APIs define the core Java platform for general-purpose computing. These APIs are in modules whose names start with java.

JDK

Figure 5.1a Java API website—part A

©Oracle/Java


Overview (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/index.html
OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Java SE 12 & JDK 12
SEARCH:
nextint
Members

java.util.Scanner.hasNextInt()
java.util.Scanner.hasNextInt(int)
java.util.PrimitiveIterator.OfInt.nextInt()
java.util.Random.nextInt()
java.util.Scanner.nextInt()
java.util.SplittableRandom.nextInt()
java.util.concurrent.ThreadLocalRandom.nextInt()
java.util.Random.nextInt(int)
java.util.Scanner.nextInt(int)
java.util.SplittableRandom.nextInt(int)

Java® Platform, Standard E
Version 12 API Specificatio

This document is divided into two sections

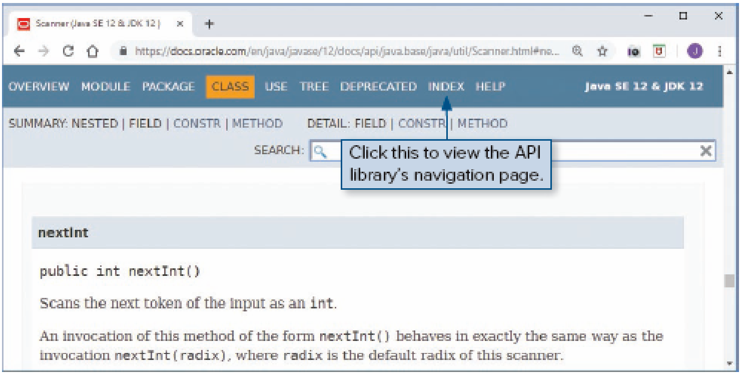
Java SE

The Java Platform, Standard E
for general-purpose computing
with java.

Figure 5.1b Java API website—part B

©Oracle/Java

Figure 5.1c shows what happens on the Java API website if you type “nextint” in the search box and click the java.util.Scanner.nextInt() entry. As you can see, you’re taken to a description of the nextInt method—how to call it, and what it does. We’ll have more to say later on about how to read such descriptions.


Scanner (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/util/Scanner.html#ne...
OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Java SE 12 & JDK 12
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH:
Click this to view the API library’s navigation page.

nextInt

`public int nextInt()`

Scans the next token of the input as an `int`.

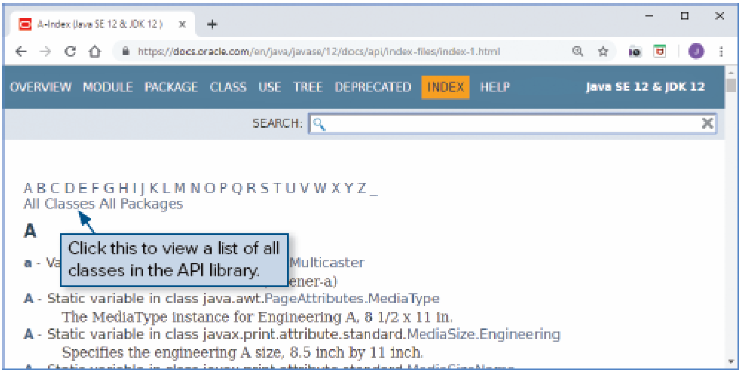
An invocation of this method of the form `nextInt()` behaves in exactly the same way as the invocation `nextInt(radix)`, where `radix` is the default radix of this scanner.

Figure 5.1c Java API website—part C

©Oracle/Java

If you hope that the API library contains a class that might help with your current programming project, but you're unsure, you might want to peruse the list of all classes and see if anything jumps out at you. To see the Java API class list, click the INDEX tab (as shown in Figure 5.1c), which displays the navigation page shown in Figure 5.1d, and on that page, click the All Classes link.

Figure 5.1e shows the Java API class list. There are thousands of API library classes, but fear not, the list is alphabetized, and most classes have multiword names where the first word is the same for a group of classes. When you see the ten classes that begin with "Audio," and your program doesn't use sound, skipping those methods is fast. Feel free to use the scrollbar to peruse the list of classes that appears at the left or press ctrl+f (f for find) to generate a search box. Figure 5.1f shows the result when you press ctrl+f, type “math” in the search box, and click on the found Math class in the list of classes. As you can see, you’re taken to a description of the Math class. If you know your program needs to do some fancy math, scrolling down through the Math class’s methods might reveal something useful.

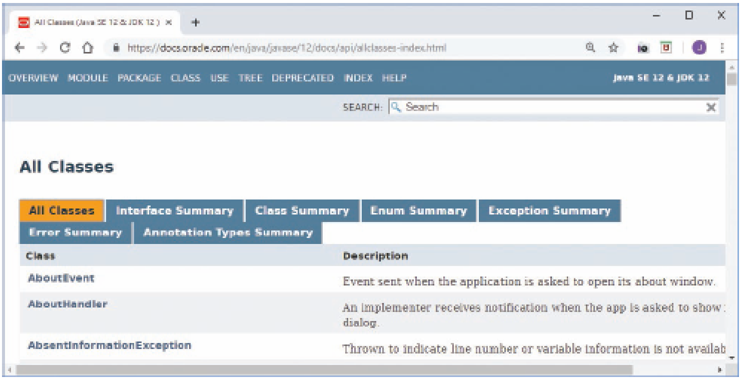

A-Index (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/index-files/index-1.html
OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Java SE 12 & JDK 12
SEARCH:
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z _
All Classes
All Packages
A
a - Va
Multicaster
ener-a)

Click this to view a list of all classes in the API library.

A - Static variable in class java.awt.PageAttributes.MediaType
    The MediaType instance for Engineering A, 8 1/2 x 11 in.
A - Static variable in class javax.print.attribute.standard.MediaSize.Engineering
    Specifies the engineering A size, 8.5 inch by 11 inch.

Figure 5.1d Java API website—part D

©Oracle/Java


All Classes (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/allclasses-index.html
OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Java SE 12 & JDK 12
SEARCH:
Search
All Classes
All Classes
Interface Summary
Class Summary
Enum Summary
Exception Summary
Error Summary
Annotation Types Summary

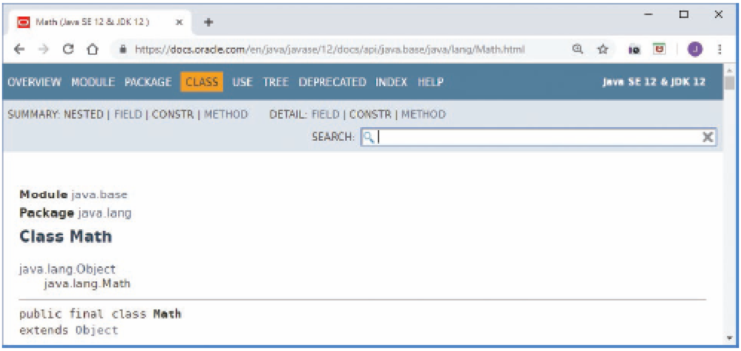
Class	Description
AboutEvent	Event sent when the application is asked to open its about window.
AboutHandler	An implementer receives notification when the app is asked to show : dialog.
AbsentInformationException	Thrown to indicate line number or variable information is not availab

Figure 5.1e Java API website—part E

©Oracle/Java

With so many classes (thousands), Oracle attempts to organize them into groups. A group of classes is known as a *package*. To view Java's standard packages, click on the MODULE tab at the top of the Java API web page. There, you'll see the list of packages in the java.base module, which contains most of the standard packages used in this book. The concept of a Java module is rather advanced, and we postpone a full description until Appendix 4. For now, just think of a module as another grouping mechanism—a module is a group of packages. If you scroll down on the web page that shows the list of packages, you can see the first couple of packages – java.io (which we describe in the Files chapter later on) and java.util (which we've been using for input for a while now). Figure 5.1g shows what that web page looks like with those two packages listed.

Using Oracle's Java API website is like surfing the web, but you're not surfing the whole world. You're just surfing the Java API library. You can do it, and we encourage you to give it a try whenever you're curious.


Math (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/java.base/java/lang/Math.html
OVERVIEW MODULE PACKAGE CLASS USE TREE DEPRECATED INDEX HELP
Java SE 12 & JDK 12
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH:

Module java.base
Package java.lang

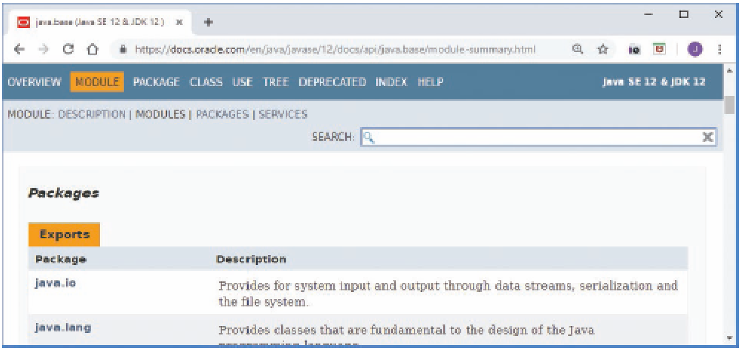
Class Math

java.lang.Object
    java.lang.Math

```
public final class Math
extends Object
```

Figure 5.1f Java API website—part F

©Oracle/Java


java.base (Java SE 12 & JDK 12)
https://docs.oracle.com/en/java/javase/12/docs/api/java.base/module-summary.html
OVERVIEW
MODULE
PACKAGE
CLASS
USE
TREE
DEPRECATED
INDEX
HELP
Java SE 12 & JDK 12
MODULE: DESCRIPTION | MODULES | PACKAGES | SERVICES
SEARCH:

Packages

Exports

Package	Description
java.io	Provides for system input and output through data streams, serialization and the file system.
java.lang	Provides classes that are fundamental to the design of the Java

Figure 5.1g Java API website—part G

©Oracle/Java

Using the API Class Library

To use an API class in your program, you must first import it (i.e., load it) into your program. For example, to use the Scanner class, you must include this statement at the top of your program:

```
import java.util.Scanner;
```

Note the java.util part of java.util.Scanner. The java.util part is the name of a package. The "util" stands for "utility," and the java.util package contains general-purpose utility classes. The only java.util class you'll need right now is the Scanner class. But there are many other useful classes in the java.util package. Examples are:

- The Random class, for helping you work with random numbers—discussed in an optional section at the end of this chapter.
- The Calendar class, for helping you work with times and dates—discussed in an optional section at the end of Chapter 8.
- The Arrays, ArrayList, LinkedList, and Collections classes, for helping you work with lists or collections of similar data—discussed in Chapters 9 and 10.

If you have a program that needs to use more than one of the classes in a particular package, like two or more of the util package classes just mentioned, you can import them all at once using a statement like this:

```
import java.util.*;
```

The asterisk is a *wildcard*. In the above statement, the asterisk causes all classes in the java.util package to be imported—not just the Scanner class. There’s no inefficiency in using the wildcard notation. The compiler includes only as much as it needs in the compiled program.

Several classes are so important that the Java compiler automatically imports them for you. These automatically imported classes are in the java.lang package, where lang stands for “language.” In effect, the Java compiler automatically inserts this statement at the top of every Java program:

```
import java.lang.*;
```

Because this is automatic and understood, there’s no need to write it explicitly.

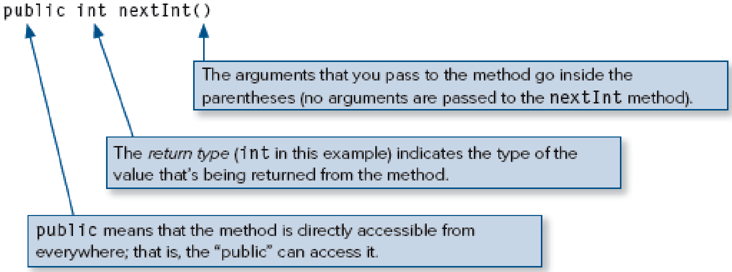
The Math class is in the java.lang package, so there's no need for you to import the Math class if you want to perform math operations. Likewise, the System class is in the java.lang package, so there's no need for you to import the System class if you want to perform a System.out.println command.

Headings for API Methods

To use an API class, you don't need to know the internals of the class; you just need to know how to "interface" with it. To interface with a class, you need to know how to use the methods within the class. For example, to perform input, you need to know how to use the Scanner class's methods—next, nextLine, nextInt, nextDouble, and so on. To use a method, you need to know what type of *arguments* to pass to it and what type of value it *returns*.

Arguments are the input you supply to a method when you call it, or ask it to do something for you, and the value it returns is the answer it gives you back.

The standard way to present method-interface information is to show the method's source code heading. For example, here's the source code heading for the Scanner class's nextInt method:



```
public int nextInt()
```

The arguments that you pass to the method go inside the parentheses (no arguments are passed to the `nextInt` method).

The *return type* (`int` in this example) indicates the type of the value that's being returned from the method.

`public` means that the method is directly accessible from everywhere; that is, the "public" can access it.

In the above nextInt heading, the public access modifier should look familiar because your main method headings all use public. We'll discuss private methods in Chapter 8. They're accessible only from within the class that defines them. Note that the nextInt method returns an int value and that it has no arguments inside the parentheses. Here's an example of a Java statement that shows how you might call the nextInt method:

```
int days = stdIn.nextInt();
```

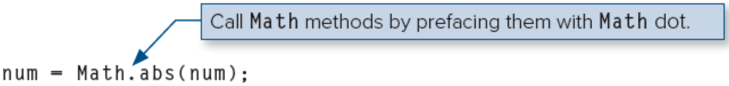
5.3 Math Class

The Math class is one of the prebuilt classes in the always available java.lang package. This class contains methods that implement standard mathematical *functions*. A mathematical function generates a numeric value based on one or more other numeric values. For example, a square root function generates the square root of a given number. Likewise, the Math class's sqrt method returns the square root of a given number. In addition to providing mathematical methods, the Math class provides two mathematical constants—π (the ratio of a circle's circumference to its diameter) and e (the base of natural logarithms).

Basic Math Methods

Let's now look at some of the Math class's methods. Throughout the book, when there's a need to present a group of methods from the API library, we'll introduce the methods by showing a list of method headings and associated brief descriptions. Headings for API methods are commonly referred to as *API headings*. Figure 5.2 contains API headings for some of the more popular methods in the Math class, with associated brief descriptions.

As you read through Figure 5.2, we hope that you'll find most of the methods to be straightforward. But some items may need clarification. Note the static modifier at the left of all the Math methods. All the methods in the Math class are static. The static modifier means they are static methods, and they must be called by prefacing the method's name with the name of the class in which they are defined. For example, here's how you'd call the abs method:



Call `Math` methods by prefacing them with `Math` dot.

```
num = Math.abs(num);
```

The above statement updates num's value, so num gets the absolute value of its original value. For example, if num starts out with −15, it ends up with 15.

Note that the following statement does not work properly:

```
Math.abs(num);
```

It finds the absolute value of num, but it does not update the content stored inside num. Math methods return a value. They do not update a value. So if you want to update a value, you must use an assignment operator.

In Figure 5.2, note that there's only one pow method, and it has two double variables (num and power) in its heading. When a variable is declared in a heading, it's referred to as a *parameter*. When it's used in a method call, it's referred to as an *argument*. There's no pow method with int parameters. But that's no big deal because you can pass an int value to the pow method. More generally, it's legal to pass an integer value to a method that accepts a floating-point argument. It's like assigning an integer value into a floating-point variable, discussed in Chapter 3. Let's see how this works within a code fragment. There is an empirical rule called "Horton's Law," which says that the length of a river scales with the area drained by the river in accordance with this formula:

$$\text{length} \approx 1.4\ (\text{area})^{0.6}$$

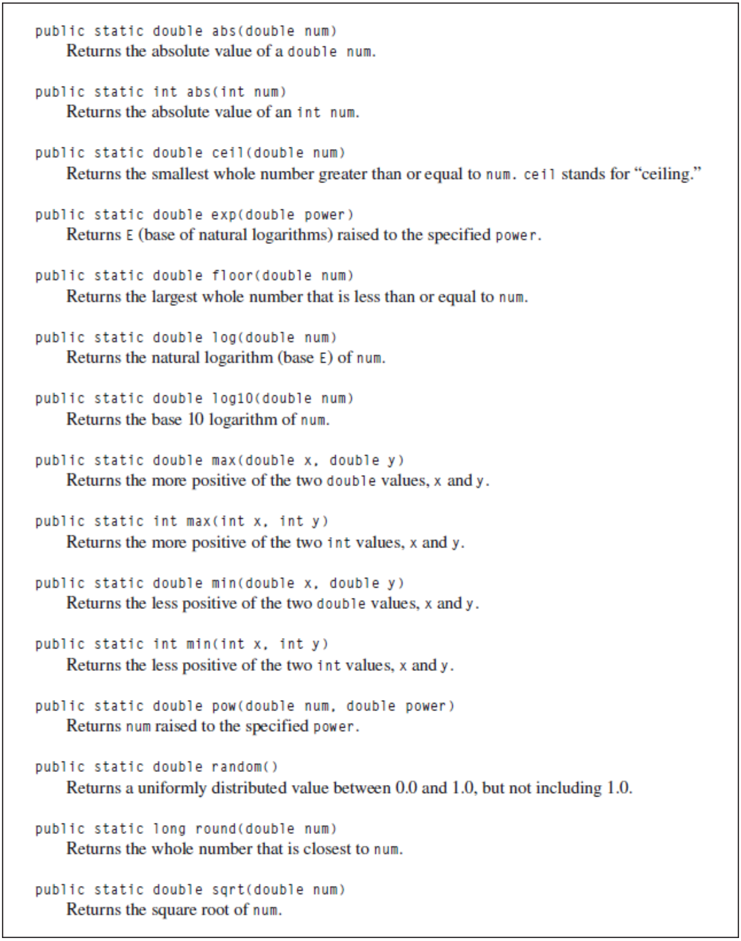
Here's how you might implement Horton's Law in Java code:

OK to pass an `int (area)` into `pow`, which accepts `double` arguments.

```
int area = 10000;       // square miles drained
System.out.println("river length = " + 1.4 * Math.pow(area, 0.6));
```

Output:

```
river length = 351.66410041134117
```



`public static double abs(double num)`
Returns the absolute value of a `double num`.

`public static int abs(int num)`
Returns the absolute value of an `int num`.

`public static double ceil(double num)`
Returns the smallest whole number greater than or equal to `num`. `ceil` stands for "ceiling."

`public static double exp(double power)`
Returns `E` (base of natural logarithms) raised to the specified `power`.

`public static double floor(double num)`
Returns the largest whole number that is less than or equal to `num`.

`public static double log(double num)`
Returns the natural logarithm (base `E`) of `num`.

`public static double log10(double num)`
Returns the base 10 logarithm of `num`.

`public static double max(double x, double y)`
Returns the more positive of the two `double` values, `x` and `y`.

`public static int max(int x, int y)`
Returns the more positive of the two `int` values, `x` and `y`.

`public static double min(double x, double y)`
Returns the less positive of the two `double` values, `x` and `y`.

`public static int min(int x, int y)`
Returns the less positive of the two `int` values, `x` and `y`.

`public static double pow(double num, double power)`
Returns `num` raised to the specified `power`.

`public static double random()`
Returns a uniformly distributed value between 0.0 and 1.0, but not including 1.0.

`public static long round(double num)`
Returns the whole number that is closest to `num`.

`public static double sqrt(double num)`
Returns the square root of `num`.

Figure 5.2 API headings and brief descriptions of some of the methods in the java.lang.Math class

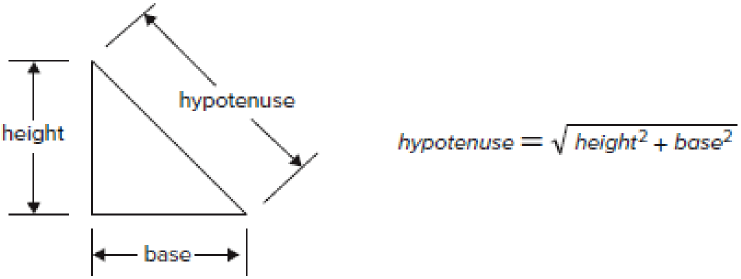
Note the round method in Figure 5.2. How is it different from using an (int) type cast operator on a double value? The (int) operator truncates the fraction, whereas the round method rounds up if the fraction is ≥0.5.

As shown in Figure 5.2, Math's random method returns a uniformly distributed value between 0.0 and 1.0, not including 1.0. "Uniformly distributed" means that there's the same chance of getting any value within the specified range. In other words, if you have a program that calls random, the chances are the same for random returning 0.317, 0.87, 0.02, or any value between 0.0 and 1.0, not including 1.0.

Why would you want to call the random method? If you need to analyze a real-world situation that involves random events, you should consider writing a program that uses the random method to model the random events. For example, if you work for a city transportation department, and you’re in charge of improving traffic flow at traffic light intersections, you could write a program that uses the random method to model the arrival of automobiles at the traffic lights. For each traffic light that you’re interested in, you’d set the traffic light’s cycle time (e.g., two minutes between each new green signal) and then simulate automobiles arriving at the traffic light at random intervals. You’d run the program so that it simulates one week of traffic flow, and you’d keep track of average wait time for all vehicles. You’d then adjust the traffic light’s cycle time (e.g., one minute and forty-five seconds between each new green signal), run the simulation again, and determine which traffic light cycle time produces shorter average wait times.

The Math.random method is handy, and we’ll use it at various points throughout the chapter. If you need to do serious work with numbers, however, you’ll want to use the API’s Random class. We describe this class later on in Section 5.8.

Let’s wrap up the discussion of Figure 5.2’s Math methods with a complete program example. Suppose you want to calculate the length of the hypotenuse of a right triangle, given the lengths of its base and height, as shown in this picture:



$$hypotenuse = \sqrt{height^2 + base^2}$$

Figure 5.3 contains a simple program that asks the user to provide base and height values. Then it uses Math’s sqrt method to calculate and print the square root of the sum of the squares. Notice that we did not use the Math.pow method to square the base and square the height. For small powers, it's more efficient just to multiply them out. 🚲

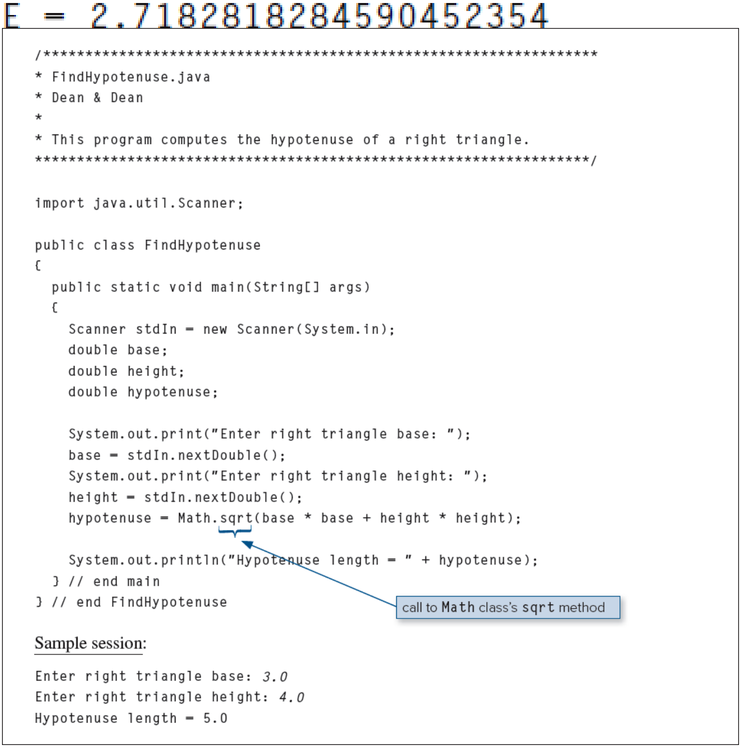
Trigonometric Math Methods

Figure 5.4 contains API headings and descriptions for some of the methods in the Math class that can help you solve problems in trigonometry. The sin, cos, and tan methods implement the sine, cosine, and tangent functions, respectively. The asin, acos, and atan methods implement the arcsine, arccosine, and arctangent functions, respectively. The trigonometric and inverse trigonometric functions all use or return angle values as radians, not degrees. Using or assuming degrees is a common programming error. Be careful! ⚠

Named Constants

The Math class also contains double values for two important named constants, π and e:

```
PI = 3.14159265358979323846

E = 2.7182818284590452354
```



```
/*********************************************************************
* FindHypotenuse.java
* Dean & Dean
*
* This program computes the hypotenuse of a right triangle.
*********************************************************************/

import java.util.Scanner;

public class FindHypotenuse
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double base;
    double height;
    double hypotenuse;

    System.out.print("Enter right triangle base: ");
    base = stdIn.nextDouble();
    System.out.print("Enter right triangle height: ");
    height = stdIn.nextDouble();
    hypotenuse = Math.sqrt(base * base + height * height);

    System.out.println("Hypotenuse length = " + hypotenuse);
  } // end main
} // end FindHypotenuse
```

call to `Math` class's `sqrt` method

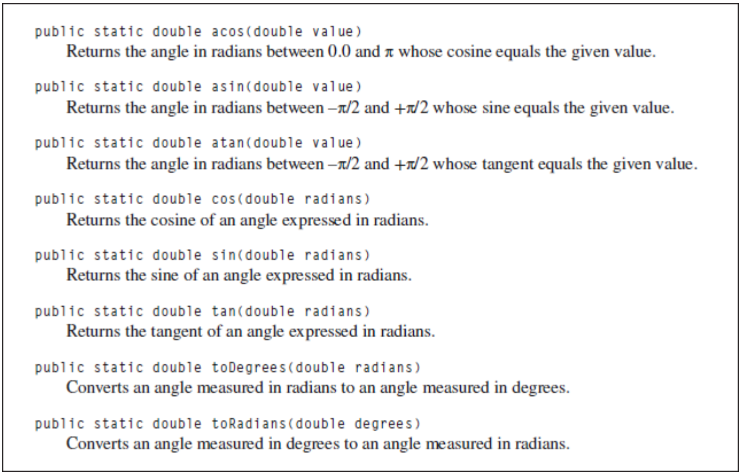
Sample session:

```
Enter right triangle base: 3.0
Enter right triangle height: 4.0
Hypotenuse length = 5.0
```

Figure 5.3 FindHypotenuse program demonstrating the use of one of Java's prebuilt math functions

PI and E are standard mathematical constants. PI is the ratio of a circle's perimeter to its diameter. E is Euler's number, the base for natural logarithm calculations. The names PI and E are in all uppercase characters, because that's standard style for named constants. Constants have fixed values, and if you attempt to assign a value to them, you'll get a compilation error. Just as Math's methods are called static methods, these constants are called *static constants,* and you access them through the Math class name. In other words, if you need π, specify Math.PI. If you look up static constants in Oracle's API, you'll see that they use the static modifier, same as for static methods, and that's why we refer to them as static constants.

Whereas we use the term "static constant" to refer to named constants that are inside Java's prebuilt classes (like the Math class), Oracle uses a different term. They use the term "constant field." In the next chapter, you'll learn about fields, but for now, just know that a field is a constant or variable that stores an object's attribute. Using the word "field" doesn't help much as a descriptor for identifying a named constant inside of a prebuilt class, so we don't use it in this context. Instead, we use the more helpful term "static constant."



`public static double acos(double value)`
Returns the angle in radians between 0.0 and π whose cosine equals the given value.

`public static double asin(double value)`
Returns the angle in radians between −π/2 and +π/2 whose sine equals the given value.

`public static double atan(double value)`
Returns the angle in radians between −π/2 and +π/2 whose tangent equals the given value.

`public static double cos(double radians)`
Returns the cosine of an angle expressed in radians.

`public static double sin(double radians)`
Returns the sine of an angle expressed in radians.

`public static double tan(double radians)`
Returns the tangent of an angle expressed in radians.

`public static double toDegrees(double radians)`
Converts an angle measured in radians to an angle measured in degrees.

`public static double toRadians(double degrees)`
Converts an angle measured in degrees to an angle measured in radians.

Figure 5.4 API headings and brief descriptions of some trigonometric methods in the java.lang.Math class

Suppose you want to compute the water needed for a 10-centimeter-diameter water balloon. Here's the - formula for the volume of a sphere:

$$\frac{\pi}{6}\,diameter^3$$

And here’s the code and resulting output for computing the volume of water for the water balloon:

```
double diameter = 10.0;
double volume = Math.PI / 6.0 * diameter * diameter * diameter;
System.out.print(”Balloon volume in cubic cm = ” + volume);
```

Output:

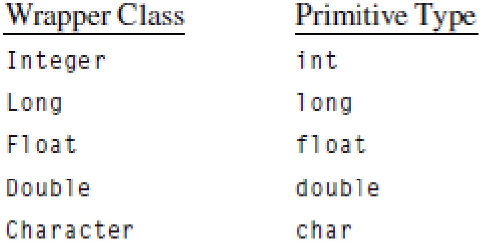
```
Balloon volume in cubic cm = 523.5987755982989
```

Some of the Math class’s static methods are extremely helpful when you need to evaluate a nontrivial mathematical function, like raising a floating-point number to a fractional power. Others do simple things you could do yourself. For example, can you think of a primitive way to do the same thing that Math.round does? It’s pretty easy. Just add 0.5 to your original double number and then use a long cast operator on that double value to end up with a rounded version of the original number. (That’s what was done in days of yore.) If it’s that easy, why bother to use Math.round? Because it makes code more readable!

The expression, Math.round(number), is self-documenting. It’s more informative than the odd-looking -expression, ((long) (0.5 + number)).

5.4 Wrapper Classes for Primitive Types

A *wrapper* is a construct that wraps (contains) a primitive data type and converts it to an object with a similar name, so it can be used in a situation where only objects are allowed. Wrapper classes do more than wrapping, however. They also provide some useful static methods and static constants. The java.lang package provides wrapper classes for all of the Java primitive types. Because this package is always available, you don’t need to use import to access these classes. Here are the wrapper classes that we’ll consider, along with the primitive types they encapsulate:



Wrapper Class	Primitive Type
`Integer`	`int`
`Long`	`long`
`Float`	`float`
`Double`	`double`
`Character`	`char`

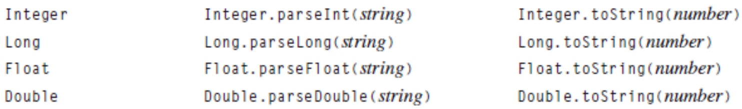
For most wrapper classes, the wrapper class’s name is the same as its associated primitive type except that it uses an uppercase first letter. There are two exceptions. The wrapper class for int is Integer, and the wrapper class for char is Character.

Methods

Like the Math class, wrapper classes contain methods and constants. We start with methods. We limit our coverage to just two sets of methods—methods that convert strings to primitives and methods that convert primitives to strings. So when would you need to convert a string to a primitive? For example, when would you need to convert the string “4” to the int 4? If you need to read a value in as a string and then later manipulate the value as a number, you’ll need to perform a string-to-number conversion. Later in this section, we’ll show a program that reads a value that could be either a number (for a lottery-number choice) or a “q” (for quitting). The program reads the user entry as a string, and if the value is not a “q”, then the program converts the user entry to a number.

Now for the other direction—when would you need to convert a primitive to a string? If you need to call a method that takes a string argument and what you’ve got is a number argument, then you’ll need to perform a number-to-string conversion. With graphical user interface (GUI) programs, all numeric output is string based. So to display a number, you need to convert the number to a string prior to calling the GUI display method. With GUI programs, all numeric input is string based, too. So to read a number, you first read the input as a string and then convert the string to a number. You'll see many examples of these processes later, in Chapters 17, 18, and 19.

Here's the syntax for converting strings to primitives and primitives to strings:



Wrapper Class	String → Primitive	Primitive → String
`Integer`	`Integer.parseInt(`*string*`)`	`Integer.toString(`*number*`)`
`Long`	`Long.parseLong(`*string*`)`	`Long.toString(`*number*`)`
`Float`	`Float.parseFloat(`*string*`)`	`Float.toString(`*number*`)`
`Double`	`Double.parseDouble(`*string*`)`	`Double.toString(`*number*`)`

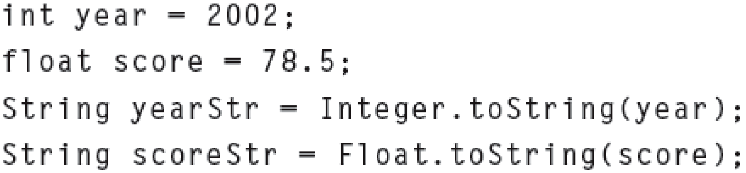
All the number wrapper classes work similarly. So if you understand how to convert from a string to an int, then you'll also understand how to convert from a string to another primitive type. To convert from a string to an int, use int's wrapper class, Integer, to call parseInt. In other words, call Integer.parseInt(*string*) and the string's corresponding int is returned.

Likewise, to convert from a string to a double, use double's wrapper class, Double, to call parseDouble. In other words, call Double.parseDouble(*string*) and the string's corresponding double is returned. Later in this section, we'll show a nontrivial example that uses the wrapper class conversion methods. But first, we'll show some trivial examples to get you used to the method-call syntax. Here we use parseInt and parseDouble to convert from strings to primitives:

```
String yearStr = "2002";
String scoreStr = "78.5";
int year = Integer.parseInt(yearStr);
double score = Double.parseDouble(scoreStr);
```

To remember the syntax for the string-to-number method calls, think of type.parse*type* for Integer.parseInt, Long.parseLong, and so on.

To convert from an int to a string, use int's wrapper class, Integer, to call toString. In other words, call Integer.toString(*int-value*) and the int value's corresponding string is returned. Likewise, to convert from a double to a string, use double's wrapper class, Double, to call toString. In other words, call Double.toString(*double-value*) and the double value's corresponding string is returned. Note this example:

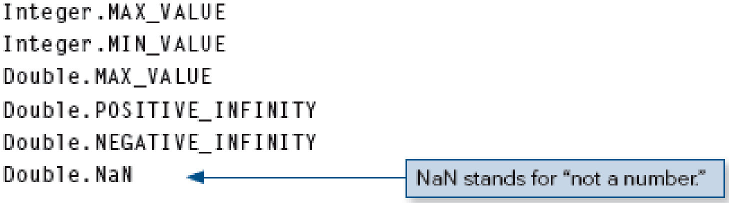


```
int year = 2002;
float score = 78.5;
String yearStr = Integer.toString(year);
String scoreStr = Float.toString(score);
```

About half the numerical wrapper class methods are static methods. We're focusing on those methods. Because they're static methods, you call them by prefacing the method call with the wrapper class's name, just as we have done.

Named Constants

The wrapper classes contain more than just methods; they also contain named constants. All the number wrappers provide named constants for minimum and maximum values. The floating-point wrappers also provide named constants for plus and minus infinity and "Not a Number," which is the indeterminate value you get if you try to divide zero by zero. Here's how you access the most important named constants defined in the Integer and Double wrapper classes:



```
Integer.MAX_VALUE
Integer.MIN_VALUE
Double.MAX_VALUE
Double.POSITIVE_INFINITY
Double.NEGATIVE_INFINITY
Double.NaN
```

NaN stands for "not a number."

There are comparable named constants for the Long and Float wrappers.

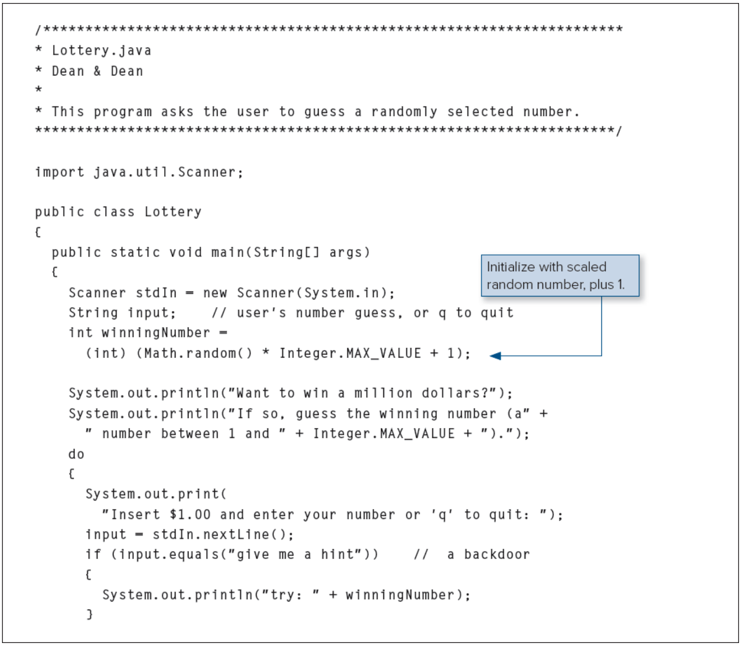
An Example

Let's put the wrapper and Math.random material into practice by showing it in the context of a complete program. Figure 5.5's Lottery program prompts the user to guess a randomly generated number between 1 and the maximum int value. The user pays $1 for each guess and wins $1,000,000 if the guess is correct. The user enters a “q” to quit.

In the initialization of winningNumber, note how the program generates a random winning-number value:

```
winningNumber = (int) (Math.random() * Integer.MAX_VALUE + 1);
```

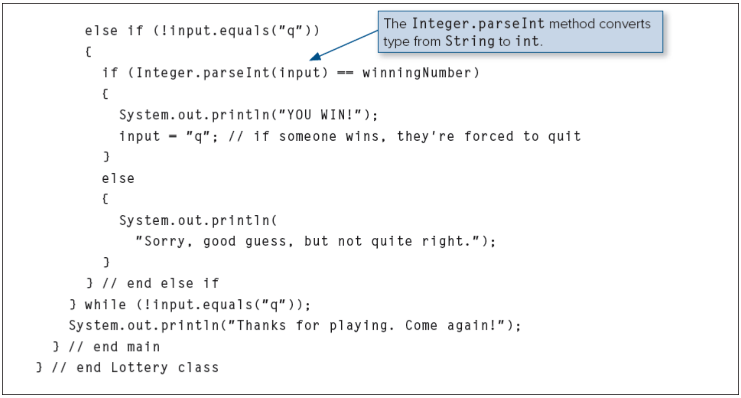
The starting point in the above equation is the call to Math.random(), which returns a random number between 0.0 and 1.0, not including 1.0. The goal is to return a value between 1 and the maximum int value, so the range is expanded by multiplying Math.random()’s value by Integer.MAX_VALUE. However, Math.random() returns a value less than 1. Therefore, the product of Math.random() and Integer.MAX_VALUE generates a number less than Integer.MAX_VALUE.



```
/*********************************************************************
* Lottery.java
* Dean & Dean
*
* This program asks the user to guess a randomly selected number.
*********************************************************************/

import java.util.Scanner;

public class Lottery
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String input;    // user's number guess, or q to quit
    int winningNumber =
      (int) (Math.random() * Integer.MAX_VALUE + 1);

    System.out.println("Want to win a million dollars?");
    System.out.println("If so, guess the winning number (a" +
      " number between 1 and " + Integer.MAX_VALUE + ").");
    do
    {
      System.out.print(
        "Insert $1.00 and enter your number or 'q' to quit: ");
      input = stdIn.nextLine();
      if (input.equals("give me a hint"))    //  a backdoor
      {
        System.out.println("try: " + winningNumber);
      }
```

Initialize with scaled random number, plus 1.

Figure 5.5a Lottery program illustrating use of the Integer wrapper class—part A



```
      else if (!input.equals("q"))
      {
        if (Integer.parseInt(input) == winningNumber)
        {
          System.out.println("YOU WIN!");
          input = "q"; // if someone wins, they're forced to quit
        }
        else
        {
          System.out.println(
            "Sorry, good guess, but not quite right.");
        }
      } // end else if
    } while (!input.equals("q"));
    System.out.println("Thanks for playing. Come again!");
  } // end main
} // end Lottery class
```

The `Integer.parseInt` method converts type from `String` to `int`.

Figure 5.5b Lottery program—part B

Adapt existing software to your needs

To ensure that the range of generated numbers includes Integer.MAX_VALUE, we add 1 to the generated product. For example, suppose the generated product is 2,147,483,646.33 (a number very near Integer.MAX_VALUE's 2,147,483,647, which is the large end of the possible range of numbers). Adding 1 to it produces 2,147,483,647.33. As another example, suppose the generated product is 0.26 (a number very near 0, which is the small end of the possible range of numbers). Adding 1 to it produces 1.26. The resulting sum is a floating-point number in the range from 1.0 to slightly greater than Integer.MAX_VALUE. However, the result needs to be an integer in the range from 1 to Integer.MAX_VALUE. For that result, the fractional part of the sum needs to be removed. And how can you remove the fractional part?

Use the (int) cast operator (as shown in the code fragment).

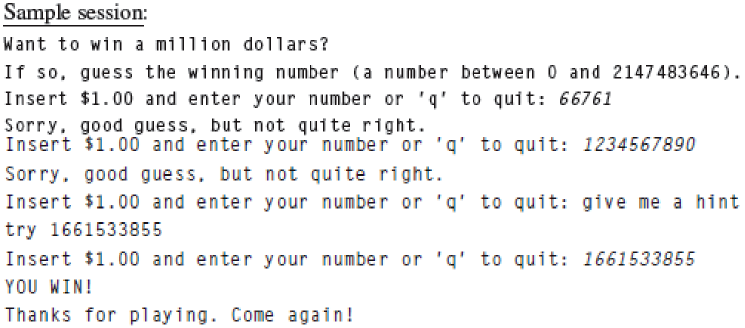
Note how the program reads in the user's number guess as a string:

```
input = stdIn.nextLine();
```

By reading the number guess as a string rather than a number, the program can handle the user entering a nonnumerical input, such as “q” for quit or “give me a hint” for a hint. If the user enters “q” the program quits. If the user enters “give me a hint,” the program prints the winning number. Great hint, eh? In this case, the hint is really a *backdoor*. A backdoor is a secret technique for gaining access to a program. The Lottery program's backdoor can be used for testing purposes.

If the user does not enter “q” or “give me a hint,” the program attempts to convert the user entry to a number by calling Integer.parseInt. The program then compares the converted number to the winning number and responds accordingly.

The Lottery program might produce the following output:



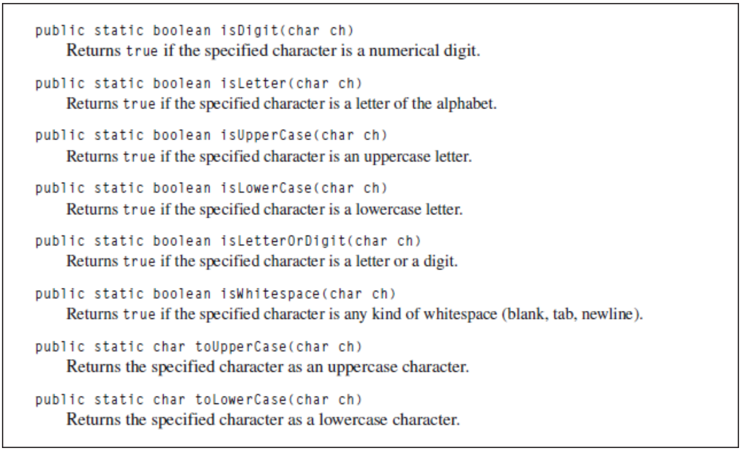
Sample session:

```
Want to win a million dollars?
If so, guess the winning number (a number between 0 and 2147483646).
Insert $1.00 and enter your number or 'q' to quit: 66761
Sorry, good guess, but not quite right.
Insert $1.00 and enter your number or 'q' to quit: 1234567890
Sorry, good guess, but not quite right.
Insert $1.00 and enter your number or 'q' to quit: give me a hint
try 1661533855
Insert $1.00 and enter your number or 'q' to quit: 1661533855
YOU WIN!
Thanks for playing. Come again!
```

5.5 Character Class

Page 186

In the previous section, we mentioned the Character wrapper class, but we didn't explain it. It's time to explain it. Often, you'll need to write programs that manipulate individual characters in a string of text. For example, you might need to read in a phone number and store just the digits, skipping the other characters (dashes, spaces, etc.). To check for digits, use the Character class's isDigit method. Figure 5.6 shows some of the more popular methods in the Character class, including the isDigit method.



`public static boolean isDigit(char ch)`
Returns `true` if the specified character is a numerical digit.

`public static boolean isLetter(char ch)`
Returns `true` if the specified character is a letter of the alphabet.

`public static boolean isUpperCase(char ch)`
Returns `true` if the specified character is an uppercase letter.

`public static boolean isLowerCase(char ch)`
Returns `true` if the specified character is a lowercase letter.

`public static boolean isLetterOrDigit(char ch)`
Returns `true` if the specified character is a letter or a digit.

`public static boolean isWhitespace(char ch)`
Returns `true` if the specified character is any kind of whitespace (blank, tab, newline).

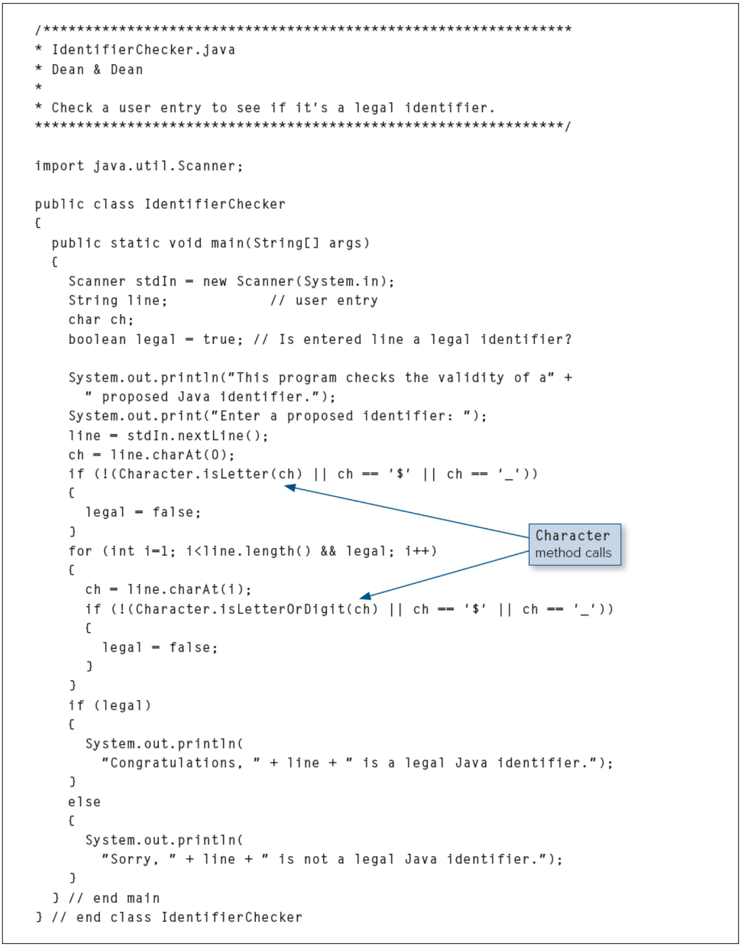
`public static char toUpperCase(char ch)`
Returns the specified character as an uppercase character.

`public static char toLowerCase(char ch)`
Returns the specified character as a lowercase character.

Figure 5.6 API headings and brief descriptions of some of the methods in the Character class

The IdentifierChecker program in Figure 5.7 illustrates the Character class in the context of a complete program. It uses the Character class's isLetter and isLetterOrDigit methods to check whether the user entry is a legal identifier.

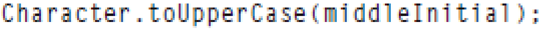
Most of Figure 5.6's methods are straightforward, but the toUpperCase and toLowerCase methods may need some clarification. Because the two methods are so similar, we'll clarify only one of them, toUpperCase. If you call toUpperCase and pass in a lowercase letter, the method returns the uppercase version of the lowercase letter. But what if you call toUpperCase and pass in an uppercase letter or a nonletter? The method returns the passed-in character, unchanged. And what if you pass in a char variable to toUpperCase instead of a char constant? The method returns the uppercase version of the passed-in char variable, but it does not change the passed-in variable's value.



```
/*******************************************************************
* IdentifierChecker.java
* Dean & Dean
*
* Check a user entry to see if it’s a legal identifier.
*******************************************************************/

import java.util.Scanner;

public class IdentifierChecker
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String line;            // user entry
    char ch;
    boolean legal = true; // Is entered line a legal identifier?

    System.out.println(”This program checks the validity of a” +
      ” proposed Java identifier.”);
    System.out.print(”Enter a proposed identifier: ”);
    line = stdIn.nextLine();
    ch = line.charAt(0);
    if (!(Character.isLetter(ch) || ch == '$' || ch == '_'))
    {
      legal = false;
    }
    for (int i=1; i<line.length() && legal; i++)
    {
      ch = line.charAt(i);
      if (!(Character.isLetterOrDigit(ch) || ch == '$' || ch == '_'))
      {
        legal = false;
      }
    }
    if (legal)
    {
      System.out.println(
        ”Congratulations, ” + line + ” is a legal Java identifier.”);
    }
    else
    {
      System.out.println(
        ”Sorry, ” + line + ” is not a legal Java identifier.”);
    }
  } // end main
} // end class IdentifierChecker
```

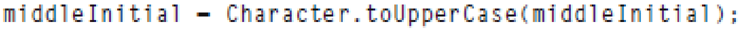
Figure 5.7 IdentifierChecker program

As evidenced by the static modifiers in Figure 5.6, most of the Character methods are static methods. Because they’re static methods, you call them by prefacing the method call with the wrapper class’s name. Let’s look at an example. Suppose you have a char variable named middleInitial and you'd like to have its content be converted to an uppercase letter. Here's a first attempt at changing middleInitial's content to an uppercase letter:



```
Character.toUpperCase(middleInitial);
```

⚠ That statement compiles and runs, but it does not change middleInitial's content. Here's the proper way to do it:



```
middleInitial = Character.toUpperCase(middleInitial);
```

5.6 String Methods

The String class is another one of the classes in the always-available java.lang package. In Chapter 3, you saw several examples of useful methods associated with objects of the String class, such as the charAt method, the length method, the equals method, and the equalsIgnoreCase method. In this section, we describe some additional String methods—the String methods shown in Figure 5.8. These String methods do not have the static access modifier, so they are not static methods, and you cannot access them with the class name. They are instance methods, and you must access them with a particular string instance. Or, said another way, you must access them with a calling-object string.

Lexicographical Ordering of Strings

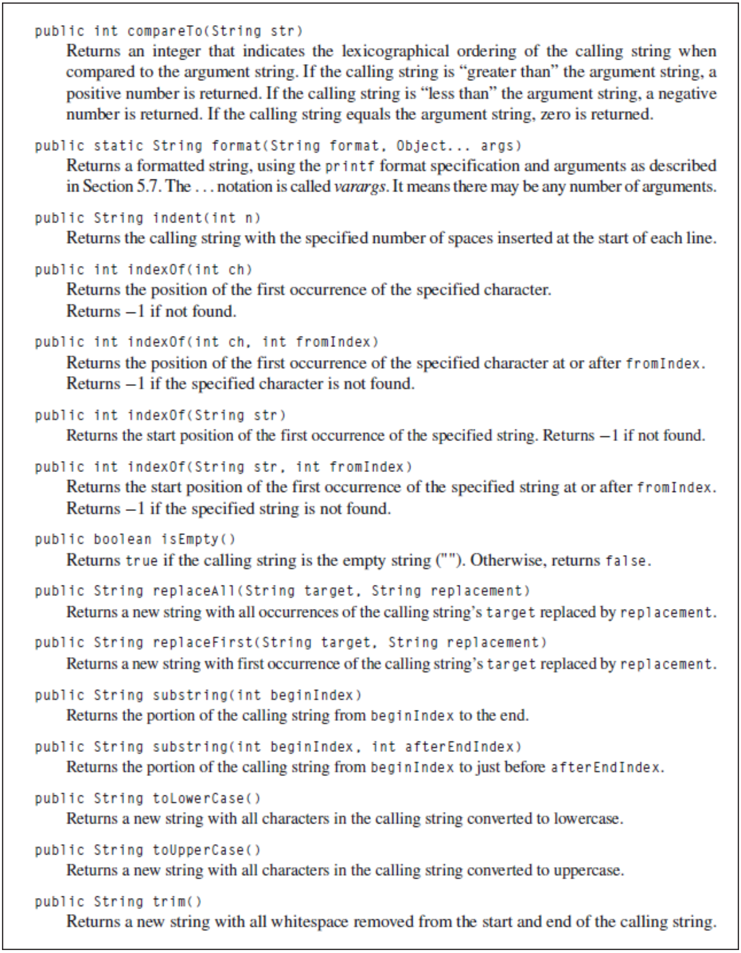
You know that numbers can be compared to determine which number is greater. Strings can also be compared. When computers compare strings to determine which string is greater, they use *lexicographical ordering*. For the most part, lexicographical ordering is the same as dictionary order. The string "hyena" is greater than the string "hedgehog" because *hyena* comes after *hedgehog* in the dictionary.

The String class’s compareTo method compares two strings to determine which is greater. As explained in Figure 5.8, compareTo returns a positive number if the calling string is greater than the argument string, a negative number if the calling string is less than the argument string, and zero if the calling string and argument string are the same. The following code fragment illustrates what we’re talking about. It compares YouTube[1] video titles and prints the results of the comparisons. If you run this code fragment, don’t be surprised if your first two output values are different from −10 and 10. According to the Java specification, the first two output values can be any negative number and any positive number, respectively.

```
String youTubeVideo = ”Colbert Super Pac”;
System.out.println(
  youTubeVideo.compareTo(”Makana We Are the Many”) + ” ” +
  youTubeVideo.compareTo(”Colbert Immigration Testimony”) + ” ” +
  youTubeVideo.compareTo(”Colbert Super Pac”));
```

Output:

```
−10 10 0
```



`public int compareTo(String str)`

Returns an integer that indicates the lexicographical ordering of the calling string when compared to the argument string. If the calling string is “greater than” the argument string, a positive number is returned. If the calling string is “less than” the argument string, a negative number is returned. If the calling string equals the argument string, zero is returned.

`public static String format(String format, Object... args)`

Returns a formatted string, using the `printf` format specification and arguments as described in Section 5.7. The . . . notation is called *varargs*. It means there may be any number of arguments.

`public String indent(int n)`

Returns the calling string with the specified number of spaces inserted at the start of each line.

`public int indexOf(int ch)`

Returns the position of the first occurrence of the specified character.
Returns –1 if not found.

`public int indexOf(int ch, int fromIndex)`

Returns the position of the first occurrence of the specified character at or after `fromIndex`.
Returns –1 if the specified character is not found.

`public int indexOf(String str)`

Returns the start position of the first occurrence of the specified string. Returns –1 if not found.

`public int indexOf(String str, int fromIndex)`

Returns the start position of the first occurrence of the specified string at or after `fromIndex`.
Returns –1 if the specified string is not found.

`public boolean isEmpty()`

Returns `true` if the calling string is the empty string (`""`). Otherwise, returns `false`.

`public String replaceAll(String target, String replacement)`

Returns a new string with all occurrences of the calling string’s `target` replaced by `replacement`.

`public String replaceFirst(String target, String replacement)`

Returns a new string with first occurrence of the calling string’s `target` replaced by `replacement`.

`public String substring(int beginIndex)`

Returns the portion of the calling string from `beginIndex` to the end.

`public String substring(int beginIndex, int afterEndIndex)`

Returns the portion of the calling string from `beginIndex` to just before `afterEndIndex`.

`public String toLowerCase()`

Returns a new string with all characters in the calling string converted to lowercase.

`public String toUpperCase()`

Returns a new string with all characters in the calling string converted to uppercase.

`public String trim()`

Returns a new string with all whitespace removed from the start and end of the calling string.

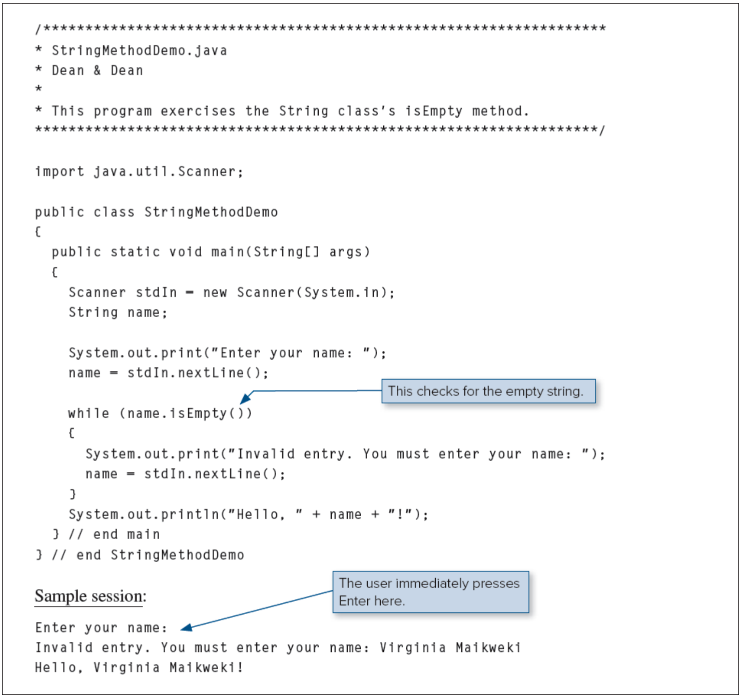
Figure 5.8 API headings and brief descriptions of some of the methods in the String class

Checking for the Empty String

Previously, you learned that the empty string is a string that contains no characters, and it's represented by two quotes with nothing between them—" ". Sometimes you'll need to check a string variable to see whether it contains the empty string. For example, when reading an input string from a user, you might want to check for the empty string as part of input validation. The following if condition illustrates:

```
if (userInput.equals(""))
```

Because checking for the empty string is such a common need, the Java API provides a method to handle that need. The isEmpty method returns true if the calling string contains the empty string and false otherwise. Figure 5.9's program uses the isEmpty method as part of an input validation while loop. The while loop forces the user to enter a non-empty name.



```
/*********************************************************************
* StringMethodDemo.java
* Dean & Dean
*
* This program exercises the String class's isEmpty method.
*********************************************************************/

import java.util.Scanner;

public class StringMethodDemo
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String name;

    System.out.print("Enter your name: ");
    name = stdIn.nextLine();

    while (name.isEmpty())
    {
      System.out.print("Invalid entry. You must enter your name: ");
      name = stdIn.nextLine();
    }
    System.out.println("Hello, " + name + "!");
  } // end main
} // end StringMethodDemo
```

This checks for the empty string.

Sample session:

```
Enter your name:
Invalid entry. You must enter your name: Virginia Maikweki
Hello, Virginia Maikweki!
```

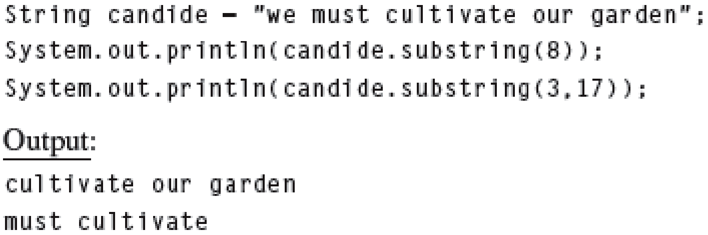
The user immediately presses Enter here.

Figure 5.9 StringMethodDemo program exercising the String class's isEmpty method

Substring Retrieval

Note the two substring methods in Figure 5.8. The one-parameter substring method returns a string that is a subset of the calling-object string, starting at the beginIndex parameter's position and extending to the end of the calling-object string. The two-parameter substring method also returns a string that is a subset of the calling-object string. The returned substring spans from the position of its first parameter, beginIndex, to the position just to the left of its second parameter, afterEndIndex.

The following code fragment processes a quote from Candide.[2] In its candide.substring(8) method call, candide is the calling object, and 8 is the beginIndex parameter value. Because string indices start at 0, the 8 refers to candide's ninth character, which is 'c'. Thus, the first println statement prints "cultivate our garden." In the candide.substring(3,17) method call, the 3 and 17 refer to candide's 4th and 18th characters, which are the '*m*' in "*must*" and the blank space character after *"cultivate."* Remember that substring's second argument indicates the position just to the right of the extracted string. Thus, the substring(3, 17) method call returns a string that spans from the ‘*m*’ in “*must*” to the ‘*e*’ in “*cultivate.*”



```
String candide - "we must cultivate our garden";
System.out.println(candide.substring(8));
System.out.println(candide.substring(3,17));
```

Output:

```
cultivate our garden
must cultivate
```

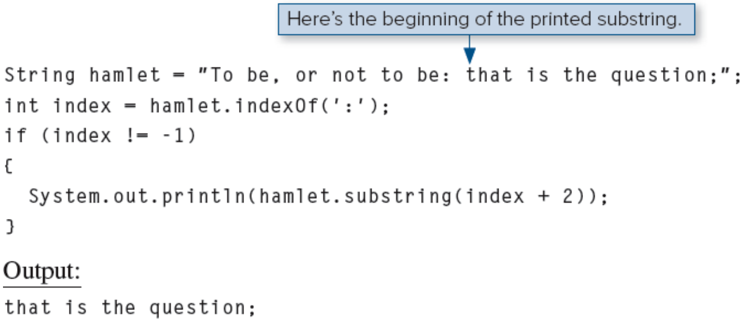
If you want to test the above code fragment or any of the following String method code fragments, use Figure 5.9’s program as a template. More specifically, replace Figure 5.9’s main method body with the new code fragment. Then compile and run the resulting program.

Position Determination

Note the one-parameter indexOf methods in Figure 5.8. They return the position of the first occurrence of a given character or substring within the calling-object string. If the given character or substring does not appear within the calling-object string, indexOf returns −1.

Note the two-parameter indexOf methods in Figure 5.8. They return the position of the first occurrence of a given character or substring within the calling-object string, starting the search at the position specified by indexOf's second parameter. If the given character or substring is not found, indexOf returns −1.

It's common to use one of the indexOf methods to locate a character or substring of interest and then use one of the substring methods to extract a nearby string. For example, consider this code fragment:[3]



Here's the beginning of the printed substring.

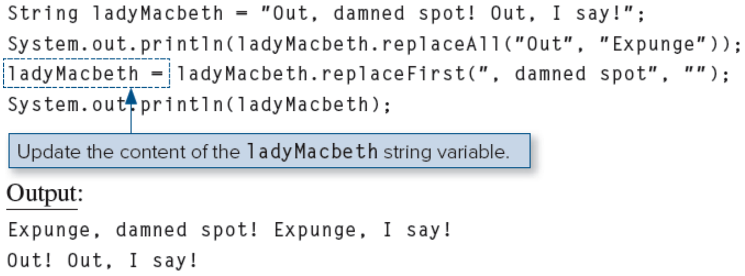
```
String hamlet = "To be, or not to be: that is the question;";
int index = hamlet.indexOf(':');
if (index != -1)
{
  System.out.println(hamlet.substring(index + 2));
}
```

Output:

```
that is the question;
```

Text Replacement

Note the replaceAll and replaceFirst methods in Figure 5.8. The replaceAll method searches its calling-object string for target, replaceAll's first parameter. It returns a new string, in which all occurrences of target are replaced with replacement, replaceAll's second parameter. The replaceFirst method works the same as replaceAll except that only the first occurrence of the searched-for target string is replaced. Here's an example that illustrates both methods:[4]



```
String ladyMacbeth = "Out, damned spot! Out, I say!";
System.out.println(ladyMacbeth.replaceAll("Out", "Expunge"));
ladyMacbeth = ladyMacbeth.replaceFirst(", damned spot", "");
System.out.println(ladyMacbeth);
```

Update the content of the `ladyMacbeth` string variable.

Output:

```
Expunge, damned spot! Expunge, I say!
Out! Out, I say!
```

Note how the second statement prints the Lady Macbeth quote with both occurrences of "Out" replaced by "Expunge", but it does not change the content of the ladyMacbeth string object. You can tell that it doesn't change the content of the ladyMacbeth string object because the next two statements generate Out! Out, I say!, where "Out" is used, not "Expunge". The reason that the second statement's replaceAll method does not change the content of the ladyMacbeth string object is that string objects are *immutable*, which means unchangeable. String methods such as replaceAll and replaceFirst return a new string, not an updated version of the calling-object string. If you really want to change the content of a string variable, you need to assign a new string object to it. That's what happens in the third statement, where the JVM assigns the result of the replaceFirst method call to the ladyMacbeth variable.

In the Lady Macbeth example, the replaceFirst method call deletes the "damned spot" by replacing it with an empty string. Because there is only one occurrence of “damned spot”, replaceAll would yield the same result as replaceFirst. But replaceFirst is slightly more efficient, and that’s why we use it here.

Whitespace Removal and Case Conversion

Note the trim, toLowerCase, and toUpperCase methods in Figure 5.8. The trim method removes all whitespace from before and after a calling-object string. The toLowerCase method returns a string identical to the calling-object string except that all the characters are lowercase. The toUpperCase method returns an uppercase version of the calling-object string. To see how these methods work, suppose we change the previous Hamlet code to this:

```
String hamlet = "To be, or not to be: that is the question;";
int index = hamlet.indexOf(':');
```

```
String hamlet2 = hamlet.substring(index + 1);
System.out.println(hamlet2);
hamlet2 = hamlet2.trim();
hamlet2 = hamlet2.toUpperCase();
System.out.println(hamlet2);
```

Now the output looks like this:

Output:

```
 that is the question;
THAT IS THE QUESTION;
```

Note how the trim method strips the leading space from hamlet2's string. Also, note how the toUpperCase method returns an all-uppercase version of hamlet2.

Insertion

To make an insertion, you must know where you want to make it. If you don't already know the index of where you want the insertion to start, you can find it by using the indexOf method with a unique substring argument. Then extract the substring up to that index, concatenate the desired insertion, and concatenate the substring after that index. The following code fragment performs two insertions within a string. More specifically, the code fragment starts with a philosophy espoused by 17th century French mathematician and philosopher René Descartes: “All nature will do as I wish it.” It then inserts two strings and transforms the message into a starkly contrasting quote from Charles Darwin: “All nature is perverse & will not do as I wish it.”[5]

```
String descartes = "All nature will do as I wish it.";
String darwin;
int index;
index = descartes.indexOf("will");
darwin = descartes.substring(0, index) +
  "is perverse & " +
  descartes.substring(index);
index = darwin.indexOf("do");
darwin = darwin.substring(0, index) +
  "not " +
  darwin.substring(index);
System.out.println(darwin);
```

Output:

```
All nature is perverse & will not do as I wish it.
```

StringBuilder and StringBuffer

When you want to compose a string out of two or more substrings, instead of using concatenation, it's more efficient in terms of memory usage and speed to use a StringBuilder or StringBuffer. Each of these classes has a constructor that takes a string as an argument. Each has an append(stringfragment) method and an insert(offset, stringfragment) method. Each also has a toString() method, but usually this is not needed, because the System.out.print and System.out.println methods automatically convert any kind of object to a String.

The following code illustrates how to use StringBuilder's insert method to shorten the previous subsection's Descartes/Darwin code fragment:

```
String descartes = "All nature will do as I wish it.";
StringBuilder darwin = new StringBuilder(descartes);
int index;
index = descartes.indexOf("will");
darwin = darwin.insert(index, "is perverse & ");
index = darwin.indexOf("do");
darwin = darwin.insert(index, "not ");
System.out.println(darwin);
```

This code generates the same output as the code in the previous subsection. This code also works and generates the same output if we substitute StringBuffer for the two instances of StringBuilder in the previous code fragment's second statement, like this:

```
StringBuffer darwin = new StringBuffer(descartes);
```

So which is better—StringBuilder or StringBuffer? In most situations, you should use the newer StringBuilder class because it's slightly faster. However, StringBuffer objects are thread-safe and StringBuilder objects are not. Thread-safe means if you have a multithreaded program, the different threads in your program will work properly and not interfere with each other. So what in the world is *multithreading*? It's when you have a program with different parts (*threads*) that can be executed concurrently in order to speed up the overall execution of the program. For a more complete explanation, see Appendix 9 Multithreading on this book's website.

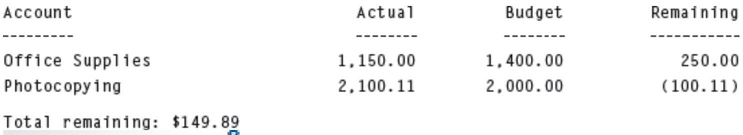
5.7 Formatted Output with the printf Method

You've used the System.out.print and System.out.println methods for quite a while now. They work fine most of the time, but there's a third System.out method that you'll want to use every now and then for formatted output. It's the printf method, where the "f" stands for "formatted." We describe the printf method in this section.

Formatted Output

For most programs, the goal is to

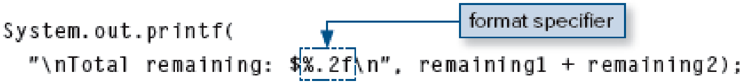
calculate something and then display the result. It’s important that the displayed result is understandable. If it’s not understandable, then no one will bother to use the program, even if it calculates flawlessly. One way to make your displayed results understandable is to format your output. By that, we mean having data columns align properly, having floating-point numbers show the same number of digits after the decimal point, and so on. Note the formatting in the budget report below. The left column is left-aligned. The other columns are right aligned. The numbers show two digits at the right of the decimal point. The numbers show commas between every third digit at the left of the decimal point. And finally, the numbers show parentheses to indicate negativeness.



```
Account                          Actual      Budget      Remaining
---------                        --------    --------    -----------
Office Supplies                  1,150.00    1,400.00         250.00
Photocopying                     2,100.11    2,000.00       (100.11)

Total remaining: $149.89
```

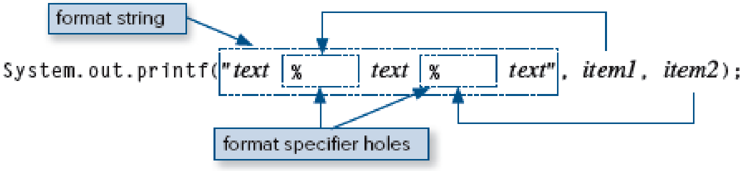
Learn how to use versatile tools.

The System.out.printf method is in charge of generating formatted output. The printf method has lots of formatting features. We'll keep things simple and explain only a few of the more popular features. We begin our explanation of the printf method by showing you how to generate the "Total remaining" line in the above budget report. Here's the code:



```
System.out.printf(
  "\nTotal remaining: $%.2f\n", remaining1 + remaining2);
```

The printf method's first argument is known as the *format string*. It contains text that prints as is, plus format specifiers that handle formatted printing. In the above example, "\nTotal remaining: $...\n" is the text that prints as is. And %.2f is the format specifier. Think of a *format specifier* as a hole where you plug in a data item. In the above example, remaining1 + remaining2 is the data item that gets plugged in. If remaining1 holds 250 and remaining2 holds −100.11, the sum is 149.89 and 149.89 gets plugged into the format specifier hole. The format specifier starts with % because all format specifiers must start with %. The format specifier's .2 causes two digits to be displayed after the decimal point. The format specifier's f indicates that the data item is a floating-point number. The example shows only one format specifier. You can have as many format specifiers as you like in a given format string. For each format specifier, you should have a corresponding data item/argument. Here's an illustration of what we're talking about:



```
System.out.printf("text %    text %    text", item1, item2);
```

Format Specifier Details

Format specifiers are powerful little critters. We won't try to describe all of their power, but we'll provide enough details to get you up and running. If you come across a formatting issue that you can't resolve with our limited coverage, look up the Formatter class on Oracle's Java API website and search for format string details. But be prepared for lots of details. The Java API provides a tremendous number of options with the printf method.

Here's the syntax for a format specifier:

```
%[flags][width][.precision]conversion-character
```

You've already seen the % symbol. It indicates the start of a format specifier. The flags, width, precision, and conversion character represent the different parts of a format specifier. Each of them specifies a different formatting trait. We'll cover them in right-to-left order. Thus, we'll describe the conversion character first. But before jumping into conversion character details, note the square brackets. They indicate that something is optional. So the flags, width, and precision parts are optional. Only the % and the conversion character are required.

Conversion Character

The conversion character tells the JVM the type of data that is to be printed. For example, it might tell the JVM to print a string, or it might tell the JVM to print a floating-point number. Here is a partial list of conversion characters:

s This displays a string.

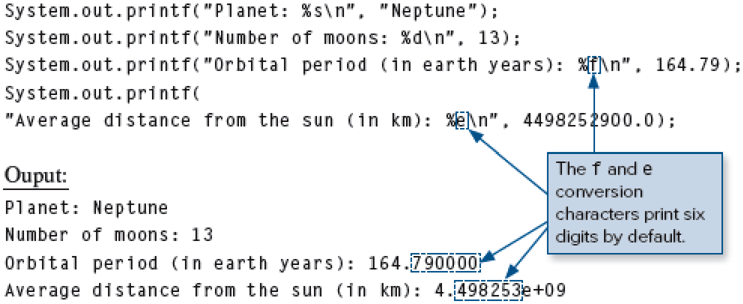
d This displays a decimal integer (an int or a long).

f This displays a floating-point

number (a float or a double) with a decimal point and at least one digit to the left of the decimal point.

e This displays a floating-point number (float or double) in scientific notation.

In explaining each part of a format specifier (conversion character, precision, width, and flags), we'll provide short examples that illustrate the syntax and semantics. After we're done with all the explanations, we'll show a complete program example. Note this code fragment and its associated output:



```
System.out.printf("Planet: %s\n", "Neptune");
System.out.printf("Number of moons: %d\n", 13);
System.out.printf("Orbital period (in earth years): %f\n", 164.79);
System.out.printf(
"Average distance from the sun (in km): %e\n", 4498252900.0);
```

Ouput:

```
Planet: Neptune
Number of moons: 13
Orbital period (in earth years): 164.790000
Average distance from the sun (in km): 4.498253e+09
```

The f and e conversion characters print six digits by default.

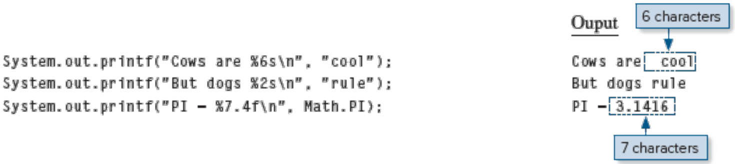
Note that by default, the f and e conversion characters generate six digits to the right of the decimal point. Also, note that if you try to print an integer value with an f or e conversion character, you'll get a runtime error.

Precision and Width

The precision part of a format specifier works in conjunction with the f and e conversion characters; that is, it works with floating-point data items. It specifies the number of digits that are to be printed to the right of the decimal point. We'll refer to those digits as the *fractional digits*. If the data item has more fractional digits than the precision's value, then rounding occurs. If the data item has fewer fractional digits than the precision's value, then zeros are added to the right so that the printed value has the specified number of fractional digits.

The width part of a format specifier specifies the minimum number of characters that are to be printed. If the data item contains more than the specified number of characters, then all the characters are printed. If the data item contains fewer than the specified number of characters, then spaces are added. By default, output values are right-aligned, so when spaces are added, they go on the left side.

Note this code fragment and its associated output:



```
System.out.printf("Cows are %6s\n", "cool");
System.out.printf("But dogs %2s\n", "rule");
System.out.printf("PI - %7.4f\n", Math.PI);
```

Ouput

```
Cows are   cool
But dogs rule
PI -  3.1416
```

In the third statement above, note the %7.4f specifier. It's easy to get fooled by the 7.4. It looks like it might be saying "seven places to the left of the decimal point and four places to the right of the decimal point," but it's actually saying "seven total spaces, with four places to the right of the decimal point." And don't forget that the decimal point is counted as one of those seven total spaces. Math.PI's value is 3.141592653589793, and when it gets printed with four places to the right of the decimal point, it gets rounded to 3.1416. ⚠

Flags

As a refresher, here's the syntax for a format specifier:

%[*flags*][*width*][*.precision*]*conversion-character*

We've described the conversion, precision, and width parts of a format specifier. It's now time to discuss flags. Flags allow you to add supplemental formatting features, one flag character for each formatting feature. Here's a partial list of flag characters:

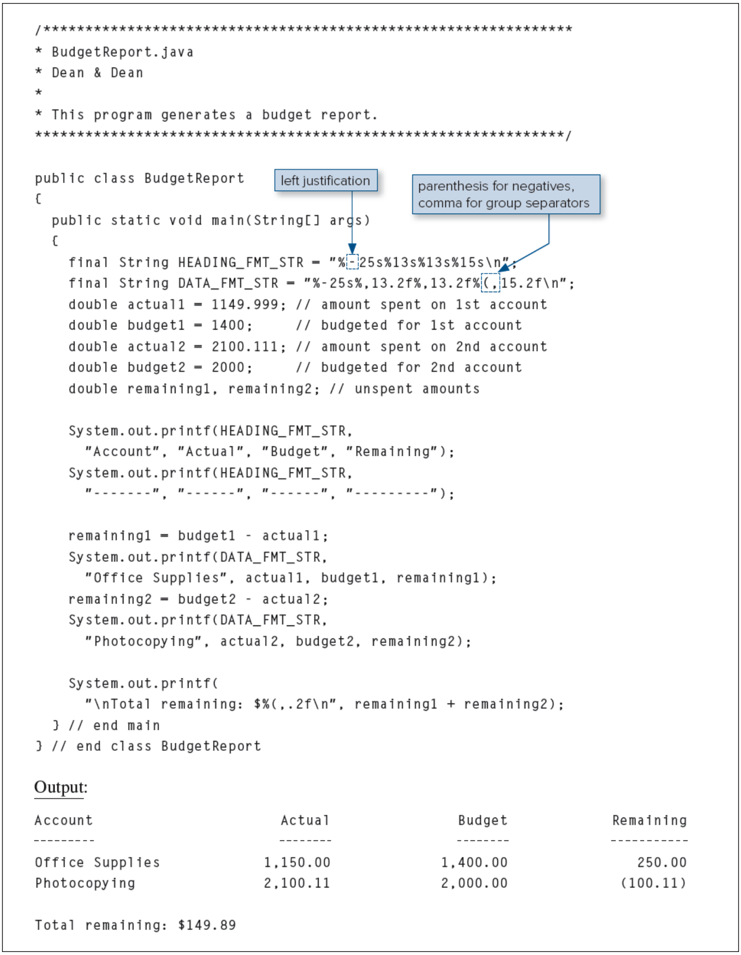
`-` Display the printed value using left justification.

`0` If a numeric data item contains fewer characters than the width specifier's value, then pad the printed value with leading zeros (i.e., display zeros at the left of the number).

, Display a numeric data item with locale-specific grouping separators. In the United States, that means commas are inserted between every third digit at the left of the decimal point.

(Display a negative numeric data item using parentheses, rather than using a minus sign. Using parentheses for negative numbers is a common practice in the field of accounting.

Let's see how format specifiers work in the context of a complete program. See Figure 5.10's BudgetReport program. Note that we use the same format string for printing the column headers and the column underlines, and the format string is stored in a HEADING_FMT_STR named constant. If you use a format string in more than one place, it’s a good idea to save the format string in a named constant and use the named constant in the printf statements. By storing the format string in one common place (in a named constant), you ensure consistency and make it easier to update the format string in the future.



```
/*********************************************************************
* BudgetReport.java
* Dean & Dean
*
* This program generates a budget report.
*********************************************************************/

public class BudgetReport
{
  public static void main(String[] args)
  {
    final String HEADING_FMT_STR = "%-25s%13s%13s%15s\n";
    final String DATA_FMT_STR = "%-25s%,13.2f%,13.2f%(,15.2f\n";
    double actual1 = 1149.999; // amount spent on 1st account
    double budget1 = 1400;     // budgeted for 1st account
    double actual2 = 2100.111; // amount spent on 2nd account
    double budget2 = 2000;     // budgeted for 2nd account
    double remaining1, remaining2; // unspent amounts

    System.out.printf(HEADING_FMT_STR,
      "Account", "Actual", "Budget", "Remaining");
    System.out.printf(HEADING_FMT_STR,
      "-------", "------", "------", "---------");

    remaining1 = budget1 - actual1;
    System.out.printf(DATA_FMT_STR,
      "Office Supplies", actual1, budget1, remaining1);
    remaining2 = budget2 - actual2;
    System.out.printf(DATA_FMT_STR,
      "Photocopying", actual2, budget2, remaining2);

    System.out.printf(
      "\nTotal remaining: $%(,.2f\n", remaining1 + remaining2);
  } // end main
} // end class BudgetReport
```

left justification

parenthesis for negatives, comma for group separators

Output:

```
Account                         Actual       Budget      Remaining
---------                     --------     --------    -----------
Office Supplies               1,150.00     1,400.00         250.00
Photocopying                  2,100.11     2,000.00       (100.11)

Total remaining: $149.89
```

Figure 5.10 BudgetReport program and its ouput

Page 199

In the BudgetReport program, note the minus sign in the HEADING_FMT_STR and DATA_FMT_STR format strings. That left justifies the first column's data.

Note the commas in the DATA_FMT_STR format string. That causes locale-specific characters (commas in the United States) to appear between every third digit at the left of the decimal point. Note the left parenthesis in the DATA_FMT_STR format string. That causes negative numbers to use parentheses instead of a minus sign.

5.8 Problem Solving with Random Numbers (Optional)

This section will show you how to generate random variables with probability distributions that are different from the 0.0 to 1.0 uniform distribution assumed in a simple Math.random method call.

Using Math.random to Generate

Random Numbers with Other Probability Distributions

As indicated in Figure 5.2, in Section 5.3, when you need a random number, you can use the Math.random method to generate one. Suppose you want a random number from a range that's different from the range 0.0 to 1.0. As we did in the initialization of winningNumber in Figure 5.5, you can expand the range to any maximum value by multiplying the random number generated by Math.random() by your desired maximum value. You can also offset the range by adding or subtracting a constant. For example, suppose you want to pick a random number that's uniformly distributed in the range between −5.0 and +15.0. Instead of using just plain old Math.random(), use this:

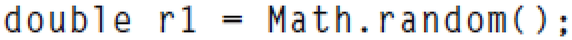
```
(20.0 * Math.random()) - 5.0.
```

It's possible to manipulate numbers

produced by Math.random to get any kind of distribution you want. For example, you can generate any of the distributions shown in Figure 5.11.

Now, let's look at how to generate these five types of random numbers from Math.random.

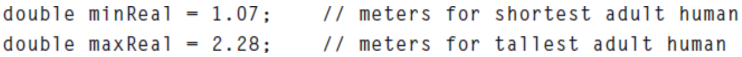
1. The first type (a continuous uniform distribution) is easy. To get a value for a random number, x, uniformly distributed in the interval between zero and unity ($0.0 \leq x \leq 1.0$), use a statement like this:



```
double r1 = Math.random();
```

   This first type of random number is the basis of all other types of random numbers.

2. For the second type (an offset and expanded continuous uniform distribution), you must have some minimum and maximum values; for example:



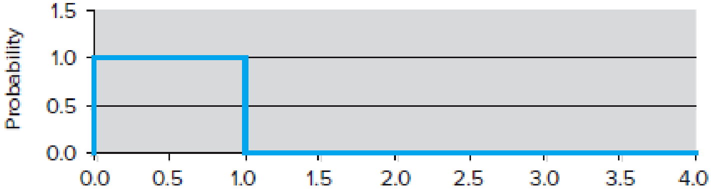
```
double minReal = 1.07;    // meters for shortest adult human
double maxReal = 2.28;    // meters for tallest adult human
```

Then you shift and expand the basic random number by using a statement like this:

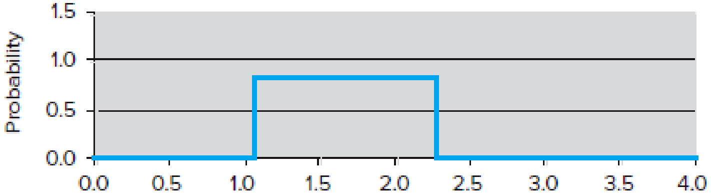
```
double r2 = minReal + Math.random() * (maxReal - minReal) ;
```

3. For the third type (a discrete uniform distribution), you create integer versions of the limits; for example:

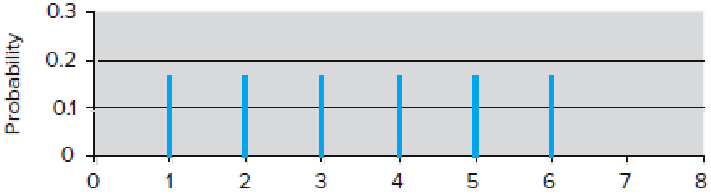
```
int min = 1;                // fewest dots on one die
int max = 6;                // most dots on one die
```



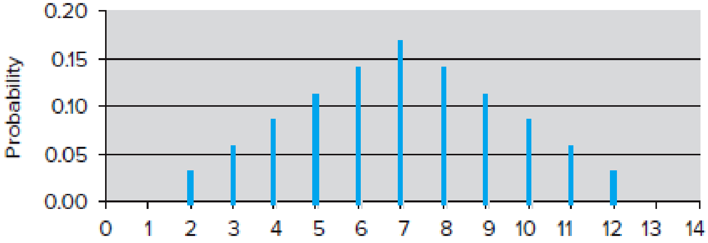
1. Random number for the default continuous uniform distribution



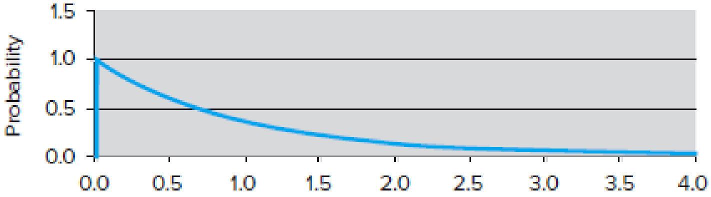
2. Random number for a scaled and offset continuous uniform distribution



3. Random number for a discrete uniform distribution



4. Random number for a discrete triangular distribution



5. Random number for a continuous exponential distribution

Figure 5.11 Important types of random

number distributions

Then you shift and expand the basic random number, sort of like you did for the second type:

```
double r3 = min + (int) (Math.random() * (max - min + 1)) ;
```

This time, you must remember that integer subtraction produces a distance that is one less than the number of integers in the range (6 minus 1 equals 5, not 6), so you have to add 1 to the difference like this (max - min + 1). The double returned by Math.random automatically promotes everything to double, so the shifted and expanded range is from 1.0 to 6.99999. The random selection gives equal weight to each of the six intervals above the integers of interest (1, 2, 3, 4, 5, and 6). The final (int) cast drops fractions.

4. For the fourth type (a discrete

triangular distribution), at first you might think you could just use the third type with min = 2 and max = 12, but that would be wrong. It would generate just as many 2’s and 12’s as 7’s, but the chance of getting a 7 is actually six times higher than getting either a 2 or a 12! The most straightforward way to get the right answer is to call Math.random twice, and add the results: ⚠

```
int twoDice = r3 + r3;
```

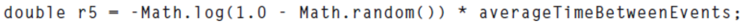
5. The fifth type of distribution (a continuous exponential distribution) has been included because it’s used in models of many important real-world phenomena, such as:

- Inter-arrival time of automobiles at an isolated traffic light
- Time between infrequent

telephone calls

- Time between radioactive emissions from an unstable atom
- Time to breakdown of a piece of machinery
- Time to failure of a semiconductor device

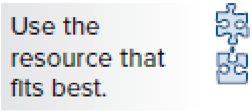
To generate a random variable with a continuous exponential distribution, use a statement like this:



```
double r5 = -Math.log(1.0 - Math.random()) * averageTimeBetweenEvents;
```

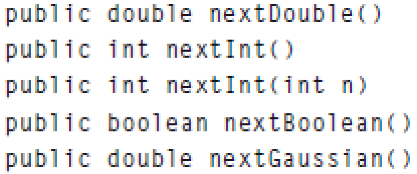
The logarithm of zero is -infinity, but that never occurs, because Math.random never generates a number as high as 1.0, so (1.0 - Math.random()) is never as low as zero.

Using the Random Class



Use the resource that fits best.

Although it is possible to get any kind of distribution from Math.random, it's not always easy. For example, the algorithm that converts Math.random's uniform distribution to a Gaussian (bell curve) distribution is rather convoluted. So it would be nice to have some prebuilt methods that immediately generate random numbers from this and other distributions. The Random class in the java.util package provides help. Here are API headings for some of the Random class's methods:



```
public double nextDouble()
public int nextInt()
public int nextInt(int n)
public boolean nextBoolean()
public double nextGaussian()
```

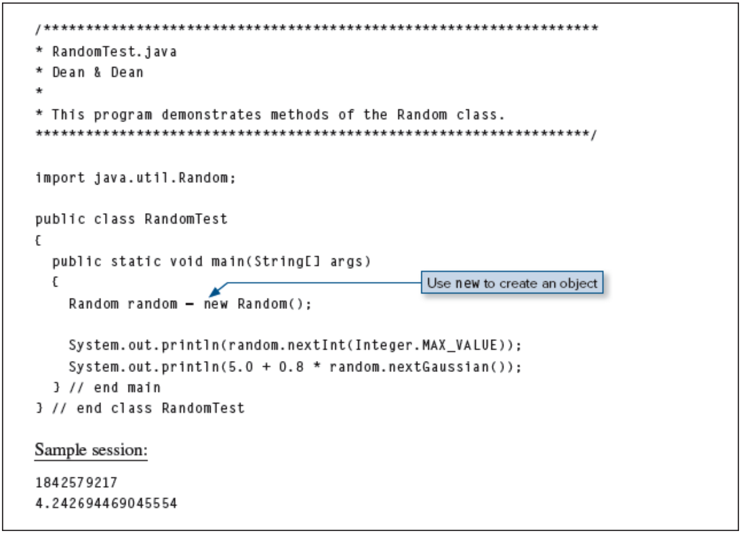
The nextDouble method does essentially the same thing as Math.random does. This distribution appears in the top graph in Figure 5.11. The zero-parameter nextInt method generates random integers uniformly from the entire range of integers, that is, from −2147483648 to +2147483647, inclusive. The one-parameter nextInt method generates random integers uniformly from zero to one less than the parameter value. This distribution is almost like what appears in Figure 5.11’s third graph for the special case of n = 7, except zero is allowed also. The nextBoolean method generates random values of true or false. The nextGaussian method generates a double value from a distribution having a mean value of 0.0 and a standard deviation of 1.0.

Notice that the Random class’s methods do not have the static modifier, so they are not static methods, and you cannot use the Random class name to access these methods. As with Scanner (described in Section 3.23), you must create an object first, and then use that object’s name to access these methods. The code in Figure 5.12 creates the object with the statement:

```
Random random = new Random();
```

This statement gives the variable called random a reference to this object. In the next two statements, this object generates two random numbers by calling the nextInt and nextGausssian methods. Due to the Integer.MAX_VALUE argument, nextInt generates a random number between 0 and one less than the maximum integer value. The nextGaussian method generates a random number drawn from a Gaussian distribution and gives it a mean of 5.0 and a standard deviation of 0.8.

When a program uses Math.random() or new Random() to generate random numbers, what appears is always a surprise because it's random! This unpredictability can be quite frustrating when you are trying to develop and test a program that uses random numbers, because every test run produces different numerical values. During development and testing, what you'd like is a repeatable sequence of "random" numbers, which turn out to be exactly the same every time you rerun a program you're testing.



```
/**********************************************************************
* RandomTest.java
* Dean & Dean
*
* This program demonstrates methods of the Random class.
**********************************************************************/

import java.util.Random;

public class RandomTest
{
  public static void main(String[] args)
  {
    Random random = new Random();

    System.out.println(random.nextInt(Integer.MAX_VALUE));
    System.out.println(5.0 + 0.8 * random.nextGaussian());
  } // end main
} // end class RandomTest
```

Use new to create an object

Sample session:

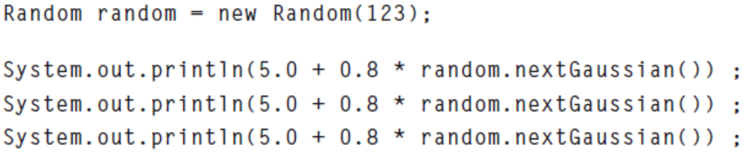
```
1842579217
4.242694469045554
```

Figure 5.12 RandomTest program using Random class's methods to generate random numbers from different distributions

Using a Fixed Seed to Freeze a

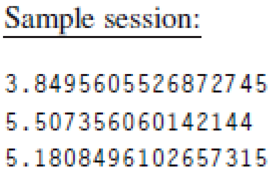
Random Number Sequence

For a repeatable random sequence, you can create a Random object with a seed. A seed provides a starting point for the internal state of the random number generator. Suppose you change the body of the main method in Figure 5.12 to this:



```
Random random = new Random(123);

System.out.println(5.0 + 0.8 * random.nextGaussian()) ;
System.out.println(5.0 + 0.8 * random.nextGaussian()) ;
System.out.println(5.0 + 0.8 * random.nextGaussian()) ;
```

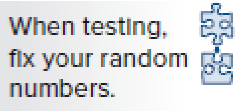
Now, if you run the program, you'll get this:



Sample session:

```
3.8495605526872745
5.507356060142144
5.1808496102657315
```

If you run the program again, and again, and again, you'll get exactly the same three "random" numbers every time! The 123 seed establishes a starting point, and this determines the "random" sequence precisely. If you pick a different seed, you'll get a different sequence, but that sequence will always be the same, as long as you stick with that particular seed. Now you know why the methods in the Random class are not static methods. They need an object to call the methods because the methods need to know some information the object contains—the seed and the current position in the random-number sequence.


When testing, fix your random numbers.

You can use the deterministic nature of the seeded random-number generator to make your life a lot easier when you are developing and debugging programs that use random numbers. To establish a fixed random-number test set, you could write a simple program that prints a particular set of random numbers. You could copy those particular numbers into assignment statements in your program—that is, hard-code them in your program for development and testing. Then, after your program has been tested and verified, you could replace each hardcoded “random number” by a random-number generator that produces a different number every time it’s invoked.

But the Random class provides a more elegant way to develop programs that have random variables. During development, create the Random object with a seed to produce exactly the same sequence of randomly distributed numbers every time that you run the program. Then, when all your bugs are fixed, simply delete the seed number to create the Random object, as in Figure 5.12, and—*voila*—your random-number generator produces completely different numbers from then on.

5.9 GUI Track: Covering

an Image with a Tinted Pane (Optional)

This section will show you how to form a composite image by covering a standard image with a semitransparent tinted pane whose coloration varies from one place to another. Figure 5.13 shows the effect.

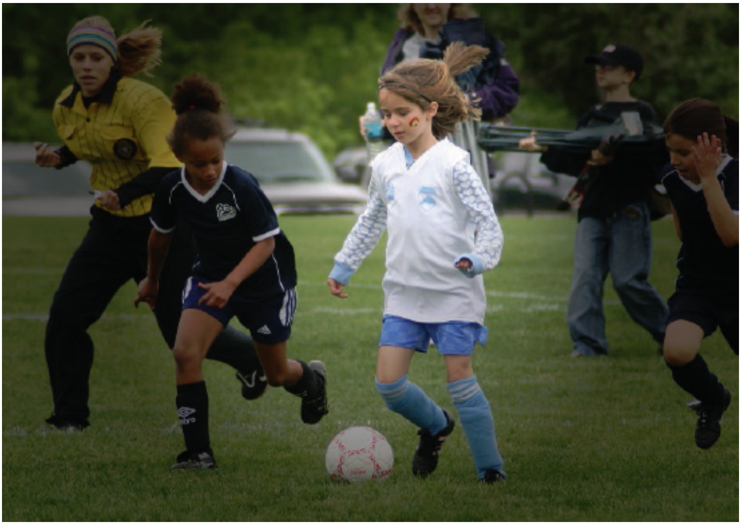


Figure 5.13 Image covered with a tinted pane, produced by program in Figure 5.14

Page 204

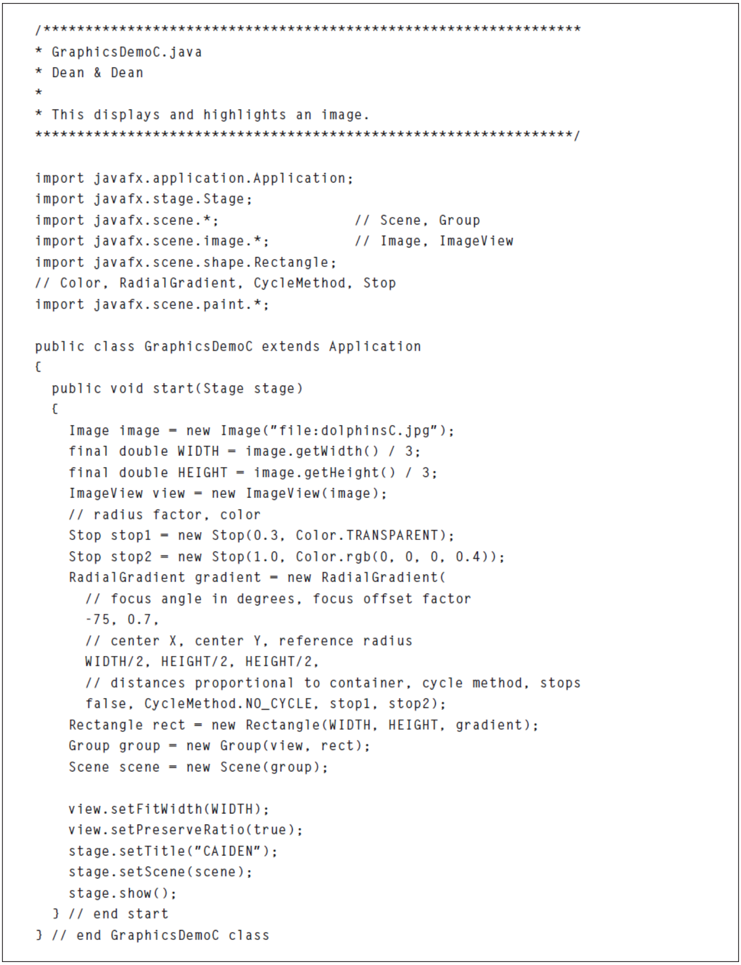
Figure 5.14 contains the program

that displays what you see in Figure 5.13. As in previous GUI programs, this program imports the JavaFX Application and Stage classes. It also imports the packages javafx.scene, javafx.scene.image, javafx.scene.shape, and javafx.scene.paint to provide access to the JavaFX classes Scene, Group, Image, ImageView, Rectangle, Color, RadialGradient, CycleMethod, and Stop.

This program's start method begins with a relatively large number of initializations that do most of its work. The first initialization assigns to the image variable an image file that's stored in the current directory (the file: indicates that a path is used rather than a URL for the file's location). The next two initializations establish pixel width and height values for the tinted pane. We scale these from the image's pixel width and height so that the tinted pane aligns properly with the transposed image underneath it.

The fourth initialization wraps the image in a special ImageView container that JavaFX is able to handle and manipulate. The next two initializations create a pair of Stop variables to be used in a subsequent RadialGradient initialization. In the Stop constructor calls, the arguments specify a radius and color. Color.TRANSPARENT means completely transparent. Color.rgb(0, 0, 0, 0.4) means no red, no green, no blue, and therefore black, and that black is 40% opaque and therefore 60% transparent.

In this case you can see that the first stop is a circle with a radius equal to 30% of the to-be-specified reference radius, and the second stop is a circle with a radius equal to the to-be-specified reference radius. The Stop object with the smallest relative radius establishes the outside of an interior region whose color matches that Stop object's color specification. With this example's CycleMethod.NO_CYCLE configuration, the Stop object with the largest relative radius establishes the inside of an exterior regime whose color matches that Stop object's color specification. If there were only one Stop object, its color would apply uniformly everywhere. With exactly two Stop objects, the color varies linearly between the circles formed by their radii. For nonlinear color variation between smallest and largest Stop objects you could insert additional Stops with relative radii of intermediate size.



```
/*******************************************************************
* GraphicsDemoC.java
* Dean & Dean
*
* This displays and highlights an image.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;                // Scene, Group
import javafx.scene.image.*;          // Image, ImageView
import javafx.scene.shape.Rectangle;
// Color, RadialGradient, CycleMethod, Stop
import javafx.scene.paint.*;

public class GraphicsDemoC extends Application
{
  public void start(Stage stage)
  {
    Image image = new Image(”file:dolphinsC.jpg”);
    final double WIDTH = image.getWidth() / 3;
    final double HEIGHT = image.getHeight() / 3;
    ImageView view = new ImageView(image);
    // radius factor, color
    Stop stop1 = new Stop(0.3, Color.TRANSPARENT);
    Stop stop2 = new Stop(1.0, Color.rgb(0, 0, 0, 0.4));
    RadialGradient gradient = new RadialGradient(
      // focus angle in degrees, focus offset factor
      -75, 0.7,
      // center X, center Y, reference radius
      WIDTH/2, HEIGHT/2, HEIGHT/2,
      // distances proportional to container, cycle method, stops
      false, CycleMethod.NO_CYCLE, stop1, stop2);
    Rectangle rect = new Rectangle(WIDTH, HEIGHT, gradient);
    Group group = new Group(view, rect);
    Scene scene = new Scene(group);

    view.setFitWidth(WIDTH);
    view.setPreserveRatio(true);
    stage.setTitle(”CAIDEN”);
    stage.setScene(scene);
    stage.show();
  } // end start
} // end GraphicsDemoC class
```

Figure 5.14 Code for GraphicsDemoC program that displays an image covered by a tinted pane

RadialGradient integrates the individual Stop objects. The Stop objects’ circles need not be concentric.

They may be offset from the center of the gradient’s reference circle in any common direction. RadialGradient’s first argument specifies that offsetting direction in degrees clockwise from right. Its second argument specifies the focus offset factor—the distance in this direction to a focus of all offsets. This focus offset factor is a 0.0 to 1.0 fraction of the reference circle’s radius. The amount of offset of a particular Stop’s center is:

stop offset distance = reference radius * focus offset factor * (1.0 – stop radius factor)

Thus, a Stop object with a zero radius factor would have its center at the location of the focus point. Our example’s stop1 is a circle with radius equal to 30% of the reference circle’s radius and centered on the current ball handler’s left collarbone. Everything inside stop1’s circle is fully transparent. Our example’s stop2 is a circle with radius equal to the radius of the reference circle and centered at the center of the image. Everything outside stop2’s circle has minimum transparency.

RadialGradient’s third and fourth arguments specify the x and y coordinates of the center of its reference circle, and the fifth argument specifies that reference circle’s radius. Since the reference circle’s radius is half the image height, the first stop’s diameter is 30% of the image height and the second stop’s diameter is the same as the image height.

RadialGradient’s sixth argument is a boolean variable that is false when the third, fourth, and fifth arguments are absolute distances in pixels, as they are in this example. Alternatively, setting the sixth argument to true changes the circle into an ellipse with horizontal and vertical radii proportional to the width and height of the RadialGradient’s container. If you use true for the sixth argument, you must change the three previous arguments (center X, center Y, and reference radius) from absolute values to fractions. To obtain a result similar to that in Figure 5.13 but with the circular Stop objects expanded horizontally into ellipses, change these three arguments to 0.5, 0.5, and 0.5, respectively. This widens the highlighted region in Figure 5.13.

RadialGradient's sixth argument, CycleMethod.NO_CYCLE, produces this example's nonrepeating gradient effect. Alternatively, the CycleMethod.REFLECT and CycleMethod.REPEAT named constants would create radial wave patterns from the focus point to the boundaries of whatever contains the RadialGradient. CycleMethod.REPEAT would create a sawtooth waveform with a wavelength in any direction equal to the distance from the focus point to the reference circle's perimeter in that direction.

CycleMethod.REFLECT would create a triangular waveform with wavelength in any direction equal to twice the distance from the focus point to the reference circle's perimeter in that direction.

The final RadialGradient arguments are any number of Stop objects.

The statement after the RadialGradient declaration puts the radial gradient into a rectangle whose width and height match the width and height of the scene in the final window. Strictly speaking, Rectangle's third argument establishes the rectangle's color. This rectangle is our "tinted pane." If this rectangle's third argument were just a simple color specification, that specification would give that rectangle a uniform tint. But there is also a more complicated kind of color—a RadialGradient!

After all this preparation, we are finally ready to set the scene in our window's stage. First, we create a JavaFX Group of two components, the previously created view (which holds the wrapped image), and the just created rect (which is the tinted pane that covers the image). Then we put the new Group into a new Scene.

The next statement scales the wrapped image pixel width down to WIDTH, which we previously set to be one-third of the original image's pixel width. The statement after that preserves that image's height-to-width ratio when the scaling occurs. The stage.setTitle method call puts the highlighted player's name (Caiden, the authors' daughter/granddaughter) at the left side of the banner at the top of the window. The stage.setScene method call puts the scene into the window, and the stage.show method call displays the window on the computer screen.

Summary

- Oracle's Java documentation identifies the public interface of all Java API software. It also provides a brief description of what it does and how to use it. The java.lang package is always available.

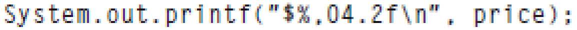
- The Math class provides methods that enable you to compute powers and roots, maximums or minimums, angle conversions, and many trigonometric functions. The random function generates a random number whose distribution is uniform in the range 0.0 to slightly less than 1.0. This class also provides named constant values for PI and E.

- Numerical wrapper classes like Integer, Long, Float, and Double contain parsing methods like parseInt, which enable you to convert String representations of numbers into numerical format. MIN_VALUE and MAX_VALUE named constants give minimum and maximum allowable values for the various numerical data types.

- The Character class provides methods that tell you whether a character is whitespace, a digit, or letter, and, if it’s a letter, whether it’s lowercase or uppercase. Other methods allow you to change case.

- The String class’s indexOf method helps you find the position of a particular character in a string of text. The substring method allows you to extract any part of a given string of text. The replaceAll and replaceFirst methods make substitutions within a string of text. You can make case conversions with the toLowerCase and toUpperCase methods, and you can use the trim method to remove whitespace from either end of a string of text.

- The first argument in the System.out.printf method is a format string which enables you to use a special code to specify the output format of text and numbers. For example, to display a double number called price as dollars and cents with commas between groups of three digits and a zero to the left of the decimal for values less than $1.00, you would write:



```
System.out.printf("$%,04.2f\n", price);
```

- Use the Random class in the java.util package to get various random-number distributions or -obtain exactly the same list of random numbers every time you run a particular program.

- Form a composite image by covering a standard image with a semitransparent tinted pane whose coloration varies from one place to another.

Review Questions

§5.3 Math Class

1. Given these declarations:

```
double diameter = 3.0;
double perimeter;
```

Provide a statement that assigns the length of a circle’s perimeter to the perimeter variable. Use the diameter variable.

2. What is the name of the class that contains the abs, min, and round methods?

a) Arithmetic

b) Math

c) Number

§5.4 Wrapper Classes for Primitive Types

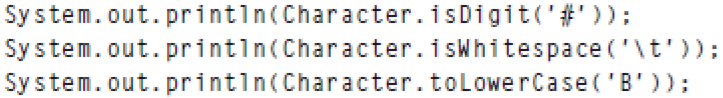
3. Provide a statement that assigns positive infinity into a double variable named num.

4. Provide a statement that converts a string variable named s to a long and assigns the result to a long variable named num.

5. Provide a statement that converts an int variable named num to a string and assigns the result to a String variable named numStr. Use a wrapper class method.

§5.5 Character Class

6. What does the following code fragment print?



```
System.out.println(Character.isDigit('#'));
System.out.println(Character.isWhitespace('\t'));
System.out.println(Character.toLowerCase('B'));
```

§5.6 String Methods

7. Given this declaration:[6]

```
String snyder = "Stick together.\nLearn the flowers.\nGo light.";
```

Write a Java statement that finds the index of the letter ‘G’ and prints everything in snyder from that point on. In other words, it prints Go light.

§5.7 Formatted Output with the printf Method

8. Write a format string that handles the display of three data items in three columns. The first column should be 20 spaces wide, and it should print a left-aligned string. The second column should be 10 spaces wide, and it should print a right-aligned integer. The third column should be 16 spaces wide, and it should print a right-aligned floating-point number in scientific format with 6 decimal places. Your format string should cause the screen's cursor to move to the next line after printing the third data item.

9. Provide a format specifier that handles the display of a floating-point data item. It should print a rounded version of the data item with no decimal places. It should insert grouping separators, and it should use parentheses if the number is negative.

§5.8 Problem Solving with Random Numbers (Optional)

10. Write a Java statement that prints a random number for the total number of dots on a roll of a pair of dice.

11. Write a program that prints five random boolean values with the seed, 123L. Then display those values.

Exercises

1. [after §5.3] Suppose you own a bond that will pay a future value based on a given annual interest rate and the number of years in the future. Here's the formula for determining the present value (what you can sell the bond for now):

$\texttt{presentValue = futureValue / (1 + rate)}^{\texttt{years}}$

Provide a Java assignment statement that calculates the present value (using future value, rate, and years), rounds to the nearest dollar, and assigns the result to a variable named presentValue. Use Math class methods.

2. [after §5.3] Dropping a pebble in a pond or striking the center of a large drum creates circular waves described by the zero-order

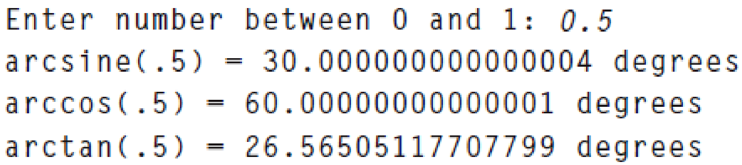
Bessel Function, denoted, $J_0(r)$, where r is a normalized radial distance. At large normalized radial distances, the zero-order Bessel Function approaches the approximation:

$J_0(r)$ → square-root(2 / (π r)) * cosine(r – π / 4)

Use Math class methods in a Java assignment statement that evaluates this large-distance approximation of J0(r).

3. [after §5.3] Write a main method that asks the user for a number between 0 and 1 and prints the arcsine, arccos, and arctan in degrees for the given number.

Sample session:



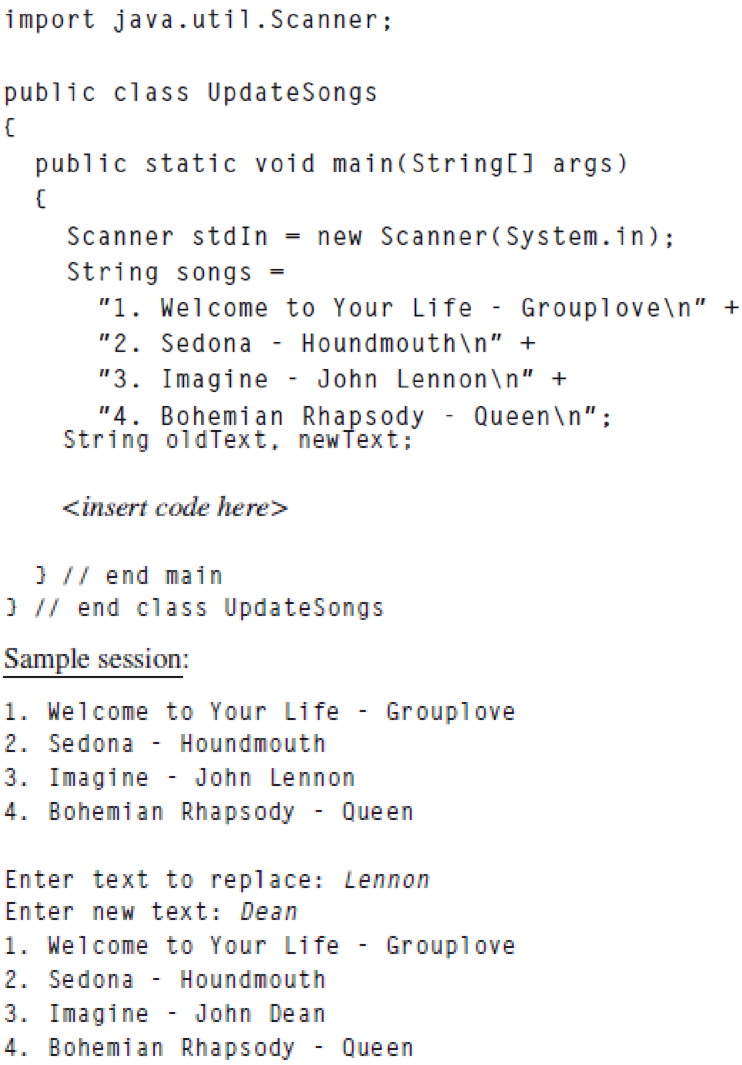
```
Enter number between 0 and 1: 0.5
arcsine(.5) = 30.000000000000004 degrees
arccos(.5) = 60.00000000000001 degrees
arctan(.5) = 26.56505117707799 degrees
```

4. [after §5.3] Imagine that you’re standing 100 meters from the base of a flagpole and the ground is flat. The flagpole is 30 meters tall. Provide a statement that prints the distance from your feet to the top of the flagpole. Use the Math class's hypot method, described on the Java API website.

5. [after §5.3] The speed of light in a vacuum is 299,792,458 meters/second. That's an exact number—not just an approximation—because the definition of a meter is based on the speed of light. Because it's an exact number you can store it in an integer. An int has 32 bits and a long has 64 bits. To know whether you can store the speed of light value in an int or a long, provide a Java statement that calculates the number of bits needed to store the speed of light (299,792,458) and assign that value into a variable named numOfBits.

The number of bits in a number x equals loge(x) / loge(2). The subscripted e is for natural logarithms. Thus, you'll want to use the Math class's natural log method.

6. [after §5.6] In the following program skeleton, replace the <*insert code here*> line with appropriate code. The resulting program should display the initial list of songs, prompt the user for a string of text to be replaced and then prompt for new text to replace it. After replacing every instance of the targeted text, it should display the updated list of songs.

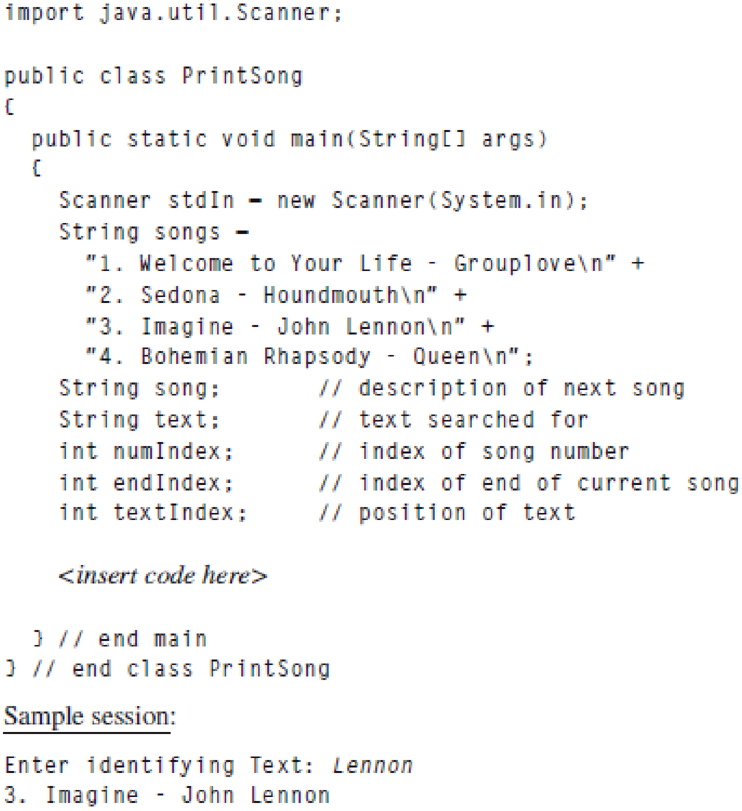


```
import java.util.Scanner;

public class UpdateSongs
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String songs =
      "1. Welcome to Your Life - Grouplove\n" +
      "2. Sedona - Houndmouth\n" +
      "3. Imagine - John Lennon\n" +
      "4. Bohemian Rhapsody - Queen\n";
    String oldText, newText;

    <insert code here>

  } // end main
} // end class UpdateSongs
```

Sample session:

```
1. Welcome to Your Life - Grouplove
2. Sedona - Houndmouth
3. Imagine - John Lennon
4. Bohemian Rhapsody - Queen

Enter text to replace: Lennon
Enter new text: Dean
1. Welcome to Your Life - Grouplove
2. Sedona - Houndmouth
3. Imagine - John Dean
4. Bohemian Rhapsody - Queen
```

7. [after §٥.٦] In the following program skeleton, replace the <*insert code here*> line with appropriate code. The resulting program should prompt the user for some identifying text, like the artist's name. Then it should display all lines in the songs string that contain that identifying text. Study the sample session.

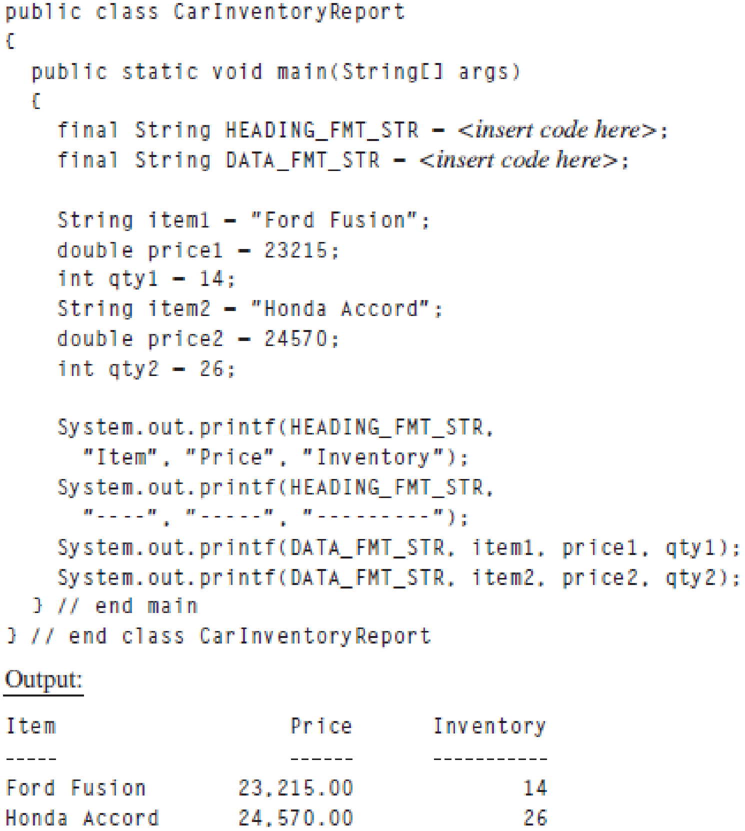


```
import java.util.Scanner;

public class PrintSong
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String songs =
      "1. Welcome to Your Life - Grouplove\n" +
      "2. Sedona - Houndmouth\n" +
      "3. Imagine - John Lennon\n" +
      "4. Bohemian Rhapsody - Queen\n";
    String song;        // description of next song
    String text;        // text searched for
    int numIndex;       // index of song number
    int endIndex;       // index of end of current song
    int textIndex;      // position of text

    <insert code here>

  } // end main
} // end class PrintSong
```

Sample session:

```
Enter identifying Text: Lennon
3. Imagine - John Lennon
```

8. [after §٥.٧] Given the below program skeleton. Replace the <*insert code here*> items with appropriate code so that the program produces the output shown below. Try to mimic the output's format precisely.



```
public class CarInventoryReport
{
  public static void main(String[] args)
  {
    final String HEADING_FMT_STR = <insert code here>;
    final String DATA_FMT_STR = <insert code here>;

    String item1 = "Ford Fusion";
    double price1 = 23215;
    int qty1 = 14;
    String item2 = "Honda Accord";
    double price2 = 24570;
    int qty2 = 26;

    System.out.printf(HEADING_FMT_STR,
      "Item", "Price", "Inventory");
    System.out.printf(HEADING_FMT_STR,
      "----", "-----", "---------");
    System.out.printf(DATA_FMT_STR, item1, price1, qty1);
    System.out.printf(DATA_FMT_STR, item2, price2, qty2);
  } // end main
} // end class CarInventoryReport
```

Output:

```
Item                  Price      Inventory
-----                ------    -----------
Ford Fusion       23,215.00             14
Honda Accord      24,570.00             26
```

9. [after §٥.٨] Given values for previousArrivalTime and averageInterArrivalTime, and assuming a continuous exponential distribution, provide Java code that uses Math.Random to generate a simulated value for nextArrivalTime.

Review Question Solutions

---

1. perimeter = Math.PI * diameter;

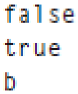
2. The class that contains the abs, min, and round methods is: b) Math

3. num = Double.POSITIVE_INFINITY;

4. num = Long.parseLong(s);

5. numStr = Integer.toString(num);

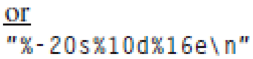
6. Here is the code fragment’s output:



```
false
true
b
```

7. System.out.println(snyder.substring(sny

8. "%-20s%10d%16.6e\n"



```
or
"%-20s%10d%16e\n"
```

(It’s OK to omit the .6 because the e conversion character prints 6 decimal places by default.)

9. "%(,.0f"

or

```
"%,(.0f"
```

(The order of flag specifier characters is irrelevant.)

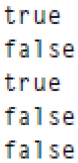
10. Statement that prints the total number of dots on a thrown pair of dice:

```
System.out.println(2 + (int) (6 * (Math.random())) +
  (int) (6 * (Math.random())));
```

11. Program that prints five random boolean values with seed 123L:

```
import java.util.Random;

public class RandomBoolean
{
  public static void main(String[] args)
  {
    Random random = new Random(123L);

    for (int i=0; i≤5; i++)
    {
      System.out.println(random.nextBoolean());
    }
  } // end main
} // end RandomBoolean
```

The values are:



```
true
false
true
false
false
```

---

[1] YouTube is a popular free video sharing website, acquired by Google in October 2006, which lets users upload, view, and share video clips.

[2] Voltaire, *Candide,* translated by Lowell Bair, Bantam Books, 1959, final sentence.

[3] Shakespeare, *Hamlet,* Act III, Sc. 1.

[4] Shakespeare, Macbeth, Act V, Sc. I.

[5] Frederick Burkhardt, ed. *Charles Darwin's Letters* (Cambridge: Cambridge University Press, 1996). Charles Darwin started college at the University of Edinburgh in 1825, studying to be a medical doctor like his father. A medical career didn't appeal to him, however, so he transferred to Cambridge University, where he

earned a B.A. in preparation for a career as a country parson. But what he really enjoyed was searching for bugs in the family barn. Right after his graduation, and before he began his first job as a country parson, family connections, a good reference from a college professor, and a pleasant personality gave him the chance to travel around the world as the companion of a brilliant sea captain named Robert FitzRoy (who later invented weather forecasting). This trip launched Darwin's career as one of the most influential scientists of the modern world.

[6] Gary Snyder, "For the Children," in *Turtle Island,* New Directions (1974).

Interlude

Multiple-Method Programs in a Non-Object-Oriented Environment

In Chapter 5, you learned how to call prebuilt methods in Oracle's Java API library. Calling prebuilt methods is an important step in your journey along the path to programming proficiency. Calling your own methods and implementing those called methods is the next logical step.

For a while now, you've been writing programs with one method—the `main` method. In this Interlude "mini-chapter," we describe how to implement multiple methods within a program and how to call the non-`main` methods from `main`. Following the description, we provide a complete program that illustrates the concepts. After that, there's a fork in the road. You'll need to learn more details about writing and calling methods, but there are two different strategies for learning those details. Many textbooks take the approach that because object-oriented programming (OOP) is such an integral part of Java programming, the additional method details should be taught in conjunction with learning OOP. On the other hand, some textbooks take a "late objects" approach and postpone a full discussion of OOP until after readers have learned the additional method details.

Each of the two approaches has its merits. The late objects approach has the advantage of being able to focus on writing methods without getting bogged down in simultaneously having to learn OOP. The other approach has the advantage of teaching good OOP habits early. Most (but not all) real-world Java programs are written using the OOP paradigm. Folks who learn with the late objects approach sometimes become enamored with non-OOP programming, to a point where it becomes harder for them to write OOP programs later on. To avoid that problem, we have a slight preference for OOP being taught in conjunction with readers learning additional method details, and that’s why we do it that way in Chapter 6. As an alternative, we provide a late objects approach, where you can learn additional method details before you learn the nuts and bolts of OOP. Most late objects advocates like to learn not only about methods before OOP, but also how to program with arrays before OOP. If you like the late objects approach, here's what we suggest you do:

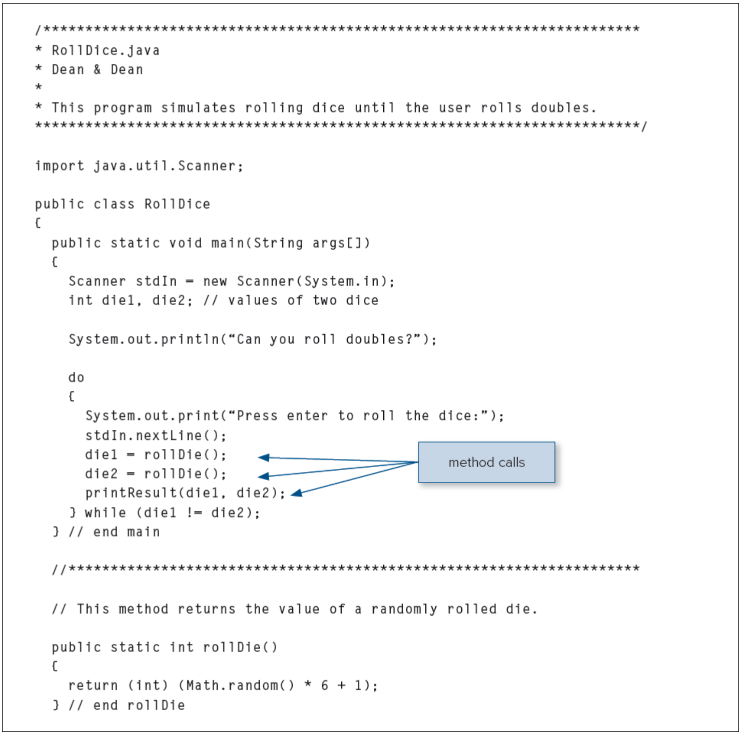
1. Read the supplemental chapter "Writing Methods in a Non-Object-Oriented Environment," which can be found at our book's resource center on Connect.

2. Read Sections 1–6 from Chapter 9 and then read the supplemental chapter "Arrays in a Non-Object-Oriented Environment," also at our book's resource center.

3. Read the rest of the book in the standard order, starting from the next chapter, Chapter 6. As part of its OOP presentation, Chapter 6 describes how to write methods that implement an object's behaviors. You should be able to cover that material quickly because it relies on many of the same concepts introduced in the "Writing Methods in a Non-Object-Oriented Environment" supplemental chapter. When you get to Chapter 9, you'll want to skip Sections 1–6 because you will have read them earlier.

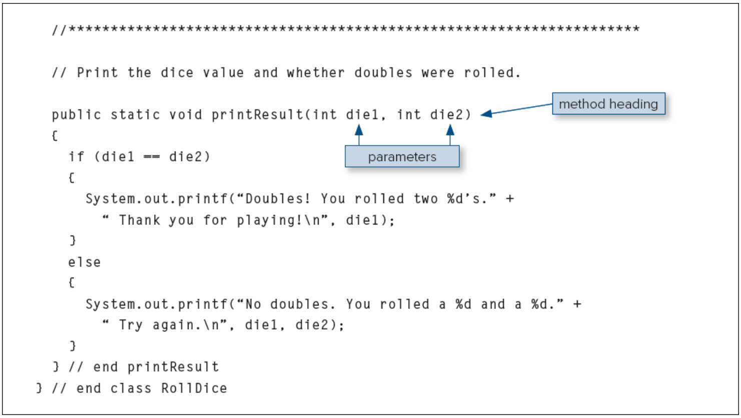
Regardless of which approach you decide to take, we encourage you to read the rest of this interlude. If you decide to go with the late objects approach, the upcoming RollDice program will serve as a teaser for what you'll see in the first supplemental chapter. If you decide to go with the standard approach, and dive into Chapter 6 right after this interlude, the RollDice program will be particularly helpful. Many real-world programs (mostly programs written in other languages, but sometimes Java programs as well) do not use the OOP paradigm. Instead, they use the *procedural programming* paradigm, in which the emphasis is on the procedures or tasks that make up a program, not the objects. The following RollDice program will serve as an example for those types of programs.

Take a look at the RollDice program shown in Figures I.1a and I.1b (the I's in the figure numbers stand for "Interlude"). The RollDice program simulates rolling two dice and determines whether "doubles" are rolled. Doubles means that the two die values are the same. So long as the die values are different, the program prompts the user to roll again.



```
/************************************************************************
* RollDice.java
* Dean & Dean
*
* This program simulates rolling dice until the user rolls doubles.
************************************************************************/

import java.util.Scanner;

public class RollDice
{
  public static void main(String args[])
  {
    Scanner stdIn = new Scanner(System.in);
    int die1, die2; // values of two dice

    System.out.println(“Can you roll doubles?”);

    do
    {
      System.out.print(“Press enter to roll the dice:”);
      stdIn.nextLine();
      die1 = rollDie();
      die2 = rollDie();
      printResult(die1, die2);
    } while (die1 != die2);
  } // end main

  //********************************************************************

  // This method returns the value of a randomly rolled die.

  public static int rollDie()
  {
    return (int) (Math.random() * 6 + 1);
  } // end rollDie
```

Figure I.1a RollDice program—part A



```
  //********************************************************************

  // Print the dice value and whether doubles were rolled.

  public static void printResult(int die1, int die2)
  {
    if (die1 == die2)
    {
      System.out.printf(“Doubles! You rolled two %d’s.” +
        “ Thank you for playing!\n”, die1);
    }
    else
    {
      System.out.printf(“No doubles. You rolled a %d and a %d.” +
        “ Try again.\n”, die1, die2);
    }
  } // end printResult
} // end class RollDice
```

Figure I.1b RollDice program—part B

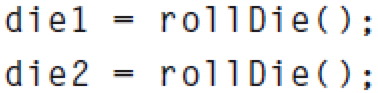
Within the RollDice program, three methods are defined—`main`, `rollDie`, and `printResult`. The entire program could have been written with just a `main` method and no other methods. That would work OK, but, as explained more fully in the upcoming chapters, whenever you have a nontrivial method, you should look for well-defined subtasks within the method. If you find such subtasks, you should consider implementing them as separate methods. In writing the RollDice program, we realized that rolling a die was a well-defined subtask, so we created a `rollDie` method. Likewise, we realized that printing the result of two dice being rolled was another well-defined subtask, so we implemented a `printResult` method. By implementing the subtasks with their own methods, we were able to keep the `main` method fairly short, and, generally speaking, short code is easier to understand than long code. Study

RollDice's `main` method in Figure I.1a. It's fairly short, and we hope that it's relatively easy to understand. Inside the loop, the program rolls each die by calling the `rollDie` method twice. The program then prints the results by calling the `printResult` method.

In the upcoming chapters, you'll learn lots of details about how to implement methods, but for now, as an introduction, let's examine the `rollDie` method briefly. Here is the single statement that forms `rollDie`'s entire body:

```
return (int) (Math.random() * 6 + 1);
```

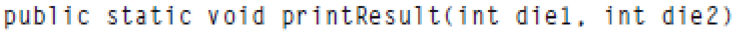
`Math.random()` generates a random floating-point number between 0.0 and 1.0, not including 1.0. Multiplying the random number by 6 generates a number between 0.0 and slightly less than 6.0. Adding 1 generates a number between 1.0 and slightly less than 7.0. Applying the `(int)` cast operator generates an integer between 1 and 6. Finally, at the left of the statement, the Java reserved word `return` causes the generated value (an integer between 1 and 6) to be passed back to the place where the method was called from. Looking at the RollDice program, you can see that `main` calls `rollDie` with these two statements:



```
die1 = rollDie();
die2 = rollDie();
```

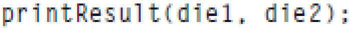
When those two statements are executed, the JVM assigns `rollDie`'s returned values to `die1` and `die2`.

Again briefly, let's examine the `printResult` method. As you can see in the RollDice program and copied below for your convenience, the `printResult` method heading declares two variables—`die1` and `die2`:



```
public static void printResult(int die1, int die2)
```

Such method heading variables are called *parameters*. They store the arguments that are passed to them from their associated method calls. In the RollDice program, you can see that `main` calls `printResult` with this statement:



```
printResult(die1, die2);
```

That method call passes its `die1` and `die2` arguments to the `die1` and `die2` parameters in the `printResult` heading. The `printResult` method then compares the `die1` and `die2` values for equality and prints a “Doubles!” or “No doubles.” message, accordingly.

There’s something else noteworthy about the `printResult` method. Do you see the word `static` in the method’s heading? As explained in Chapter 3, if you don’t use `static`, then you’ll be required to do some extra work before you can call the `printResult` method. Specifically, you’ll be required to first instantiate an object. We’ll describe instantiation in

Chapter 6. On the other hand, if you're writing a non-OOP program, then you won't be instantiating objects, and you'll need to use `static` for each method heading. Thus, in the non-OOP RollDice program, `static` appears in the `printResult` method heading and also in the `rollDie` method heading.

That's it for your brief introduction to writing multiple methods in a non-object-oriented environment. If you prefer the late objects approach, you should now jump to the first supplemental chapter on the book's -resource center and learn more about writing methods in a non-object-oriented environment. On the other hand, if you prefer the standard approach, you should move on to the next chapter, Chapter 6, and learn how to write methods in the context of an object-oriented programming environment. Either way, you'll get to the same final place. Now go forth and enjoy the journey!

GUI Track: Multiple-Method Program That Uses StackPane and Group to Display Images, Rectangles, Lines, an Oval, and Text (Optional)

This optional section shows how to transfer most of the work in a JavaFX GUI program from the `start` method to subordinate methods. This program creates a composition that includes a one-third-sized view of an original image and a highlighted full-size view of a selected area in that image. See Figure I.2. A rectangle drawn on the one-third-size view identifies the selected area. The composition includes lines that extend from the corners of the selected area to the corners of the highlighted full-sized view of the selected area. The highlighted full-sized view of the selected area is surrounded by a thicker border with slightly rounded corners, and it includes an oval name tag that identifies the selected player.

Figure I.3a contains the first part of the program that generates the display in Figure I.2. As in previous GUI programs, this program imports the JavaFX `Application` and `Stage` classes. It also imports packages that provide access to other JavaFX classes: `Scene`, `Group`, `StackPane`, `Border`, `BorderStroke`, `CornerRadii`, `Image`, `ImageView`, `Rectangle`, `Line`, `Rectangle2D`, `Insets`, `Color`, and `Text`.

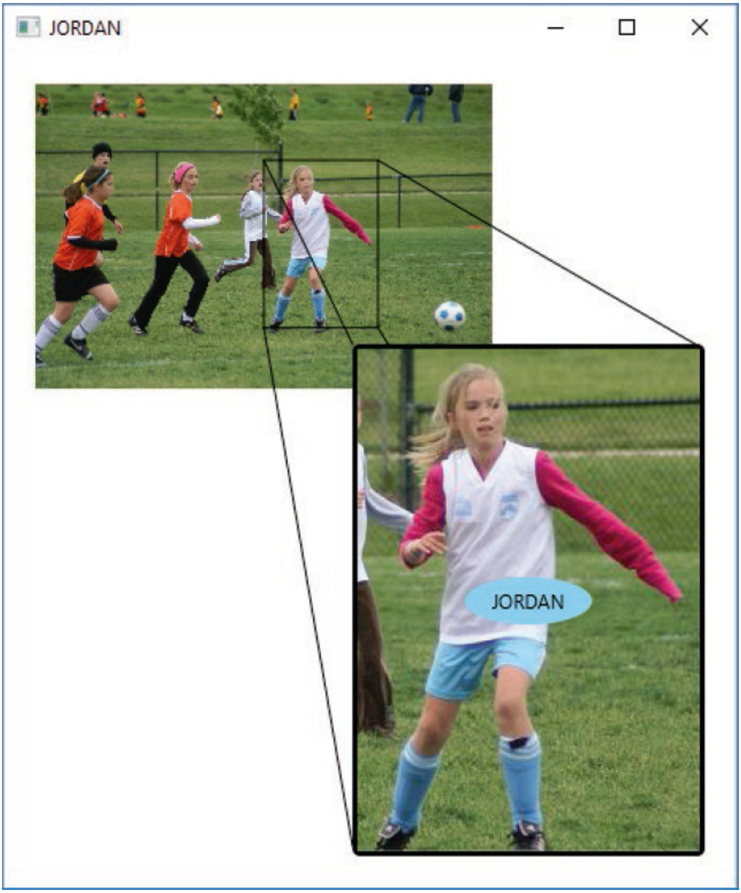
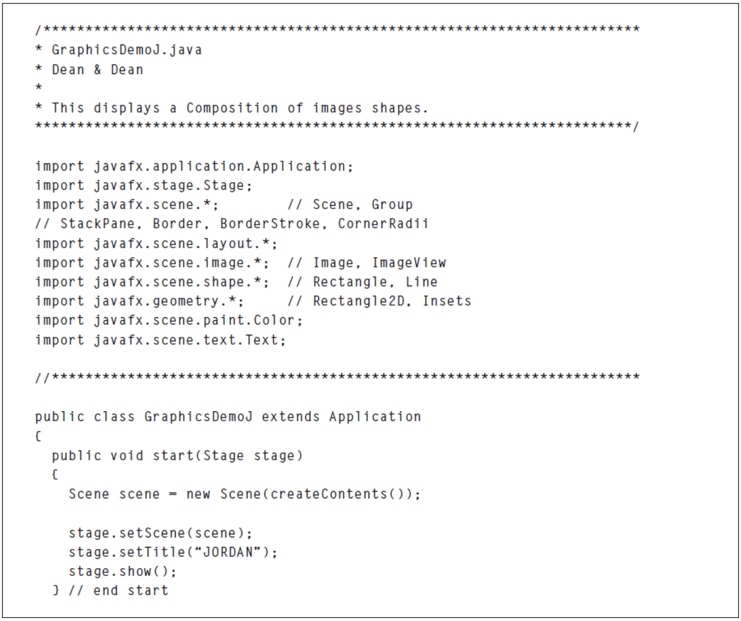


Figure I.2 Display of scaled down image and a decorated full-scale view of a selected area. The full-scale selected area has a thick round-cornered border and includes a colored oval with the selected player's name. Diagonal lines connect corresponding corners of the two rectangles.

```
/************************************************************************
* GraphicsDemoJ.java
* Dean & Dean
*
```



```
/*************************************************************************
* GraphicsDemoJ.java
* Dean & Dean
*
* This displays a Composition of images shapes.
*************************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;        // Scene, Group
// StackPane, Border, BorderStroke, CornerRadii
import javafx.scene.layout.*;
import javafx.scene.image.*;  // Image, ImageView
import javafx.scene.shape.*;  // Rectangle, Line
import javafx.geometry.*;     // Rectangle2D, Insets
import javafx.scene.paint.Color;
import javafx.scene.text.Text;

//************************************************************************

public class GraphicsDemoJ extends Application
{
  public void start(Stage stage)
  {
    Scene scene = new Scene(createContents());

    stage.setScene(scene);
    stage.setTitle(“JORDAN”);
    stage.show();
  } // end start
```

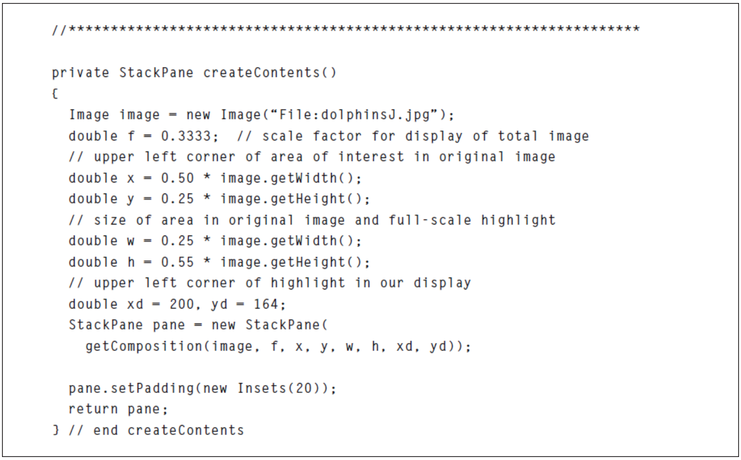
Figure I.3a GraphicsDemoJ program—part A

This program’s `start` method is relatively simple because its first statement creates the scene by calling a `createContents` helper method that returns a fully composed container. (The reserved word `new` creates a new Java object. You have seen this operation before in programs using `new Scanner`, and you will see it many times again.) The next three statements set the `scene` onto the `stage`, give the `stage` a title, and display it on the computer screen.

Figure I.3b shows the `createContents` method. It retrieves an image and establishes parameters that determine how the program will process and display that image. It copies the image from a separate file in the same directory, and it declares the parameters as local constants. (An alternative version of this `createContents` method might ask the user to supply the name of the file and/or any or all of processing parameter values.) The variable `f` is a scaling factor that specifies the size of the program's presentation of the whole image relative to the actual size of the original image. The `x`, `y`, `w`, and `h` values are pixels in the original image. The xd and yd values are pixels in our window.

The `createContents` method's last initialization statement fills a

`StackPane` container called `pane` with a `Group` container returned by another subordinate method called `getComposition`. The subsequent `pane.setPadding` method call creates the 20-pixel-wide white space you can see in Figure I.2 between the outsides of the displayed images and the scene boundary. The final statement in the `createContents` method, `return pane`, returns the now-padded composition to the `start` method. Because this finally returned object is a `StackPane`, its contents will always stay centered if the user decides to resize the window.



```
//**********************************************************************

private StackPane createContents()
{
  Image image = new Image("File:dolphinsJ.jpg");
  double f = 0.3333;  // scale factor for display of total image
  // upper left corner of area of interest in original image
  double x = 0.50 * image.getWidth();
  double y = 0.25 * image.getHeight();
  // size of area in original image and full-scale highlight
  double w = 0.25 * image.getWidth();
  double h = 0.55 * image.getHeight();
  // upper left corner of highlight in our display
  double xd = 200, yd = 164;
  StackPane pane = new StackPane(
    getComposition(image, f, x, y, w, h, xd, yd));

  pane.setPadding(new Insets(20));
  return pane;
} // end createContents
```

Figure I.3b GraphicsDemoJ program—part B

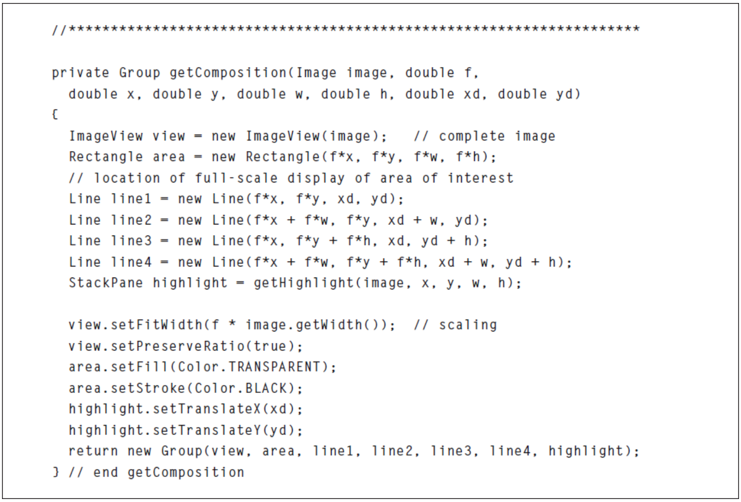
Figure I.3c contains our second subordinate method, `getComposition`. The `image` parameter is the program's copy of the overall image. The `f` parameter scales the whole image and the position and size of the rectangle that shows our area of interest. The `x` and `y` parameters specify the pixel x and y position in the original image of the upper left corner of the portion of that image we wish to feature—the full-scale view of our area of interest. The next two parameters, `w` and `h`, specify the pixel width and height of that area in the original image. The `xd` and `yd` parameters specify the pixel x and y position in the program's display of the upper left corner of the highlighted area of interest.

This `ImageView` view initialization converts the `Image` object into an

`ImageView` object, which is easier for Java to handle. The `Rectangle area` initialization creates a rectangle outlining the area of interest on our scaled down view of the original image. Each of the next four `Line` initializations creates a straight line between the point specified by its first two arguments and the point specified by its last two arguments. Each multiplication by `f` scales a number of pixels in the original image to a corresponding number of pixels in the program's display.

The `StackPane highlight` initialization calls another subordinate method, `getHighlight`. This method returns a fully populated `StackPane` container, which contains the decorated full-scale view of the area of interest.

The `view.setFitWidth` method call scales the original image width by the scaling parameter, `f`, and the subsequent `view.setPreserveRatio` method call scales its height by the same ratio. The `area.setFill` and `area.setStroke` method calls hollow out the `area` rectangle, leaving just a black border. The `highlight.setTranslateX` and `highlight.setTrnslateY` method calls specify the location in the program's display of the upper left corner of the object returned by the earlier `getHighlight` method call.



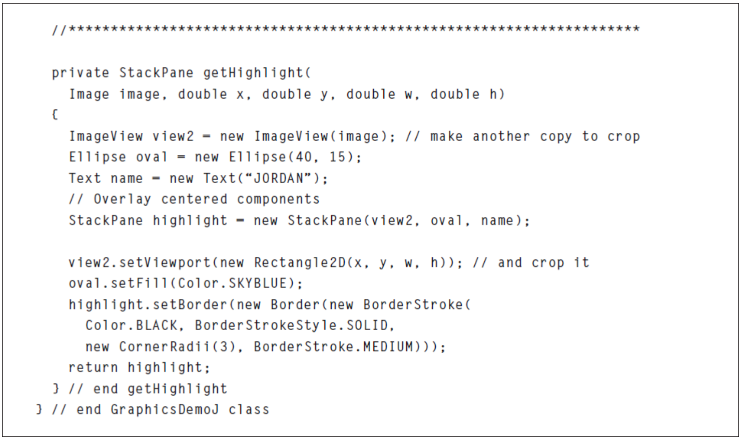
```
//********************************************************************

private Group getComposition(Image image, double f,
  double x, double y, double w, double h, double xd, double yd)
{
  ImageView view = new ImageView(image);   // complete image
  Rectangle area = new Rectangle(f*x, f*y, f*w, f*h);
  // location of full-scale display of area of interest
  Line line1 = new Line(f*x, f*y, xd, yd);
  Line line2 = new Line(f*x + f*w, f*y, xd + w, yd);
  Line line3 = new Line(f*x, f*y + f*h, xd, yd + h);
  Line line4 = new Line(f*x + f*w, f*y + f*h, xd + w, yd + h);
  StackPane highlight = getHighlight(image, x, y, w, h);

  view.setFitWidth(f * image.getWidth());  // scaling
  view.setPreserveRatio(true);
  area.setFill(Color.TRANSPARENT);
  area.setStroke(Color.BLACK);
  highlight.setTranslateX(xd);
  highlight.setTranslateY(yd);
  return new Group(view, area, line1, line2, line3, line4, highlight);
} // end getComposition
```

Figure I.3c GraphicsDemoJ program—part C

The return statement returns a `Group` container that contains everything in the scene. Like a `StackPane` a `Group` container overlays its components in the order in which they appear in its argument list. Thus, as you can see in Figure I.2, `area` overlays `view`, the four lines overlay `view`, and `highlight` overlays the four lines and `view`. However, a `Group` container has a different alignment policy. Whereas a `StackPane` automatically centers all of its components, a `Group` container expects the program to specify each component's position. If the program does not specify the position of a particular component, a `Group` container automatically positions it in its upper left corner. If you study the code in Figure I.3c, you will see that the only component it does not explicitly position is the `view` component. And sure enough, if you look at Figure I.2, you will see that the upper left corner of the `view` component is at the upper left corner of its `Group` container, which extends out to the inside of the white padding.

Figure I.3d contains our third subordinate method—the `getHighlight` method, which composes the full-sized highlight of the selected area. Its parameters are the original image and the pixel location and size of the area of interest in that image. The `ImageView view2` initialization creates a second image encapsulation that we can alter without spoiling the `ImageView view` previously created for use in our one-third scale presentation of the whole image. The `getHighlight` method will crop `view2` and save only the area of interest.

The `Ellipse oval` and `Text name` initializations create an oval and text for the name tag. The `oval` object is an instance of JavaFX's `Ellipse` class. Its first argument is `oval`'s initiation is its horizontal radius, and the second argument is its vertical radius. Both are `double` values specifying number of pixels in the display. The `text` initiation uses default font type and size. The position of an `Ellipse` object is the position of its center. The position of a `Text` object is the position of its lower-left corner. This difference in position reference makes it hard to center a text object in an oval object explicitly, so we use `StackPane` to center these items automatically. The `StackPane highlight = new StackPane(view2, oval, name)` statement creates and fills the `highlight` container by centering and overlaying the components so that `oval` covers `view2` and `name` covers `oval`.



```
  //********************************************************************

  private StackPane getHighlight(
    Image image, double x, double y, double w, double h)
  {
    ImageView view2 = new ImageView(image); // make another copy to crop
    Ellipse oval = new Ellipse(40, 15);
    Text name = new Text(“JORDAN”);
    // Overlay centered components
    StackPane highlight = new StackPane(view2, oval, name);

    view2.setViewport(new Rectangle2D(x, y, w, h)); // and crop it
    oval.setFill(Color.SKYBLUE);
    highlight.setBorder(new Border(new BorderStroke(
      Color.BLACK, BorderStrokeStyle.SOLID,
      new CornerRadii(3), BorderStroke.MEDIUM)));
    return highlight;
  } // end getHighlight
} // end GraphicsDemoJ class
```

Figure I.3d GraphicsDemoJ program—part D

The `view2.setViewport` method call uses its position and size parameters to crop the image in our second encapsulation of the original image and retain just our area of interest. The `oval.setFill` method call gives the previously created oval a sky-blue colored background. The `highlight.setBorder` method call puts a border around the centered combination of cropped image, colored oval, and name. The `BorderStroke` arguments color the border black, make it solid instead of dashed, slightly round its corners, and make its thickness somewhat greater than the default thickness used for the smaller rectangular outline that identified the area of interest in the scaled down view of the original image. The final statement returns the fully composed `highlight` object for use in the previously described `getContents` method.

Of course, we could do the same thing we have just done by moving all of the code back into the `start` method. This would seem to shorten program length by eliminating spaces between methods and reducing the number of method headers. But it would not change the total number of statements significantly. And it would make the program harder to understand. This would make it harder for anyone (including the original author!) to come back later and modify the program for a different application. In this case, for example, suppose you wanted to change the image and the location and size of the area of interest. The moral of the story is that partitioning a long program into multiple modules is almost always beneficial.

Object-Oriented Programming

Objectives

- Learn what an object is and how it relates to a class.
- Learn how to encapsulate and access data inside an object.
- Learn how to partition your

programs into “driver” and “driven” classes, to create an object of the driven class, and to give the driver a reference to that object.

- Learn the differences between an object’s data and data that is local to a method, and learn how to distinguish between those pieces of data when both have the same name.
- Understand implicit initialization (default values) of various kinds of variables.
- Learn how to trace an object-oriented program.
- Learn how to use a UML class diagram.
- Learn how to make a method return a suitable value.
- Learn how values are passed to methods.

- Write methods that get, set, and test the values of an object's data.
- Optionally learn how to improve the speed and accuracy of a simulation.

Outline

6.1 Introduction 223

6.2 Object-Oriented Programming Overview 223

6.3 First OOP Class 227

6.4 Driver Class 230

6.5 Calling Object, this Reference 234

6.6 Instance Variables 236

6.7 Tracing an OOP Program 237

6.8 UML Class Diagrams 242

6.9 Local Variables 244

6.10 The return Statement 247

6.11 Argument Passing 249

6.12 Specialized Methods—Accessors, Mutators, and Boolean Methods 252

6.13 Problem Solving with Simulation (Optional) 255

6.1 Introduction

As discussed in the Preface, we've written the book with some built-in flexibility in terms of content ordering. Readers who want an early introduction to object-oriented programming (OOP) have the option of reading Sections 6.1 through 6.8 after completing Chapter 3.

Chapter 5 served as a bridge from basic programming language constructs (variables, assignments, operators, if statements, loops, etc.) to OOP concepts. We focused primarily on one important aspect of OOP—learning how to use prebuilt methods. You used methods associated with an object, like substring and indexOf for string objects, and you used methods associated with a class, like abs and pow from the Math class. In this chapter, you'll learn how to do more than just use prebuilt classes and methods; you'll learn how to write your own classes and methods.

As you'll come to see, OOP makes large programs easier to work with. And making large programs easier to work with is very important because today's computers use lots of very large programs! The tension in learning OOP is that the first OOP programs a student can understand are necessarily small, and they can't show the power of OOP very well. But hang in there. Think of your study of this chapter and most of the next chapter as an investment. By the end of the next chapter, you'll be getting some return on that investment.

In this chapter, we start with an overview of basic OOP terms and concepts. We then step through the design and implementation of a simple OOP program. Typically, OOP design starts with a simple Unified Modeling Language (UML) class diagram, which provides a high-level, pictorial description of what you want the program to model. Then OOP design proceeds to the program's details. We'll show you how to adapt the previously described tracing technique to an OOP environment. We'll show you how to specify method details. In the previous chapter you looked at methods from the outside. Now you'll be looking at methods from the inside.

We end the chapter with an optional problem-solving section that introduces you to an important computer application—computer simulation. Computer simulation allows humans to solve problems that are difficult or impossible to solve by hand. We describe a special strategy that enables you to improve substantially both the accuracy and efficiency of computer simulations.

6.2 Object-Oriented Programming Overview

Readers who want a very early OOP overview have the option of reading this section after completing Chapter 1, Section 1.3 (Program Development).

Before OOP, the standard programming technique was *procedural programming*. Procedural programming is so named because the emphasis is on the procedures or tasks that make up a problem solution. You think first about what you want to do—your procedures. In contrast, the OOP programming paradigm invites you to think about what you want the program to represent. You typically respond to this invitation by identifying some things in the world that you want your program to model. Those things might be physical entities or conceptual entities. Once you have identified the things you want to model, you identify their basic properties/attributes. Then you determine what the things can do (their behaviors) or what the things can have done to them. You group each thing's properties and behaviors together into a coherent structure called an object. In writing an OOP program, you define objects, create them, and have them interact with each other.

Objects

An object is:

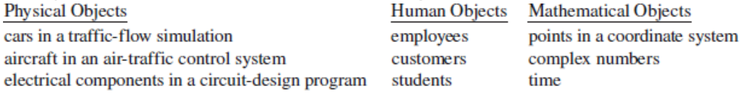
> a set of related data which identifies the current *state* of the object
>
> \+ a set of *behaviors*

Page 224

An object's state refers to the characteristics that currently define the object. For example, if you're writing a program that keeps track of employee salaries, you'd probably want to have employee objects, where an employee object's state consists of the employee's name and current salary.

An object's behaviors refer to the activities associated with the object. Once again, if you're writing a program that keeps track of employee salaries, you'd probably want to define a behavior that adjusts an employee's salary. That type of behavior parallels a real-world behavior—a pay raise or a pay cut. In Java, you implement an object's behaviors as methods. For example, you'd implement the salary adjustment behavior as an adjustSalary method. We'll describe method implementation details shortly. But it's important to complete our OOP overview first.

Here are some entities that would make good candidates for objects in an object-oriented program:



Physical Objects	Human Objects	Mathematical Objects
cars in a traffic-flow simulation	employees	points in a coordinate system
aircraft in an air-traffic control system	customers	complex numbers
electrical components in a circuit-design program	students	time

Let's think about the first example object. If a car is considered to be an object in a traffic-flow-simulation program, what is the data stored in each car object? In order to analyze traffic flow, each car's position and speed should be monitored. Therefore, those two pieces of data should be stored as part of a car object's state. And what behaviors are associated with the car objects? You'd need to be able to start the car, stop the car, slow down, and so on. So you'd probably want to implement these methods:

`start, stop, slowDown`

An object's behaviors can change an object's state. For example, a car object's start method causes the car's position and speed data items to change.

Encapsulation

Objects provide *encapsulation*. In general terms, encapsulation is when something is wrapped up inside a protective covering. When applied to objects, encapsulation means that an object's data are protected by being "hidden" inside the object. With hidden data, how can the rest of the program access an object's data? (*Accessing* an object's data refers to either reading the data or modifying it.) The rest of the program cannot access an object's data directly, but it can access the data with the help of the object's methods. Assuming that an object's methods are well written, the methods ensure that data is accessed in an appropriate manner. Returning to the employee-salaries program example, an employee object's salary should be modified only by calling the adjustSalary method. The adjustSalary method ensures that an employee object's salary is modified appropriately. For example, the adjustSalary method prevents an employee object's salary from becoming negative.

See Figure 6.1. It illustrates how an object's methods form the interface between an object's data and the rest of the program.

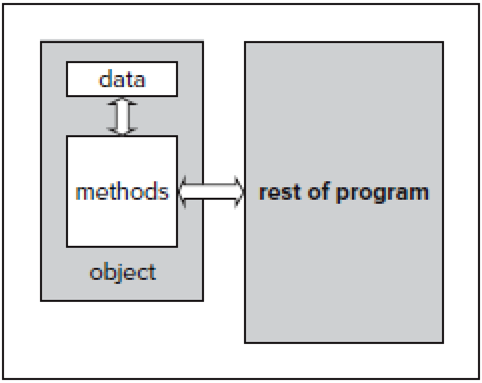


Figure 6.1 To access an object's data, you should use the object's methods as an interface

Page 225

Benefits of OOP

Now that you have a basic idea of what OOP is, you may be asking yourself what all the hype is about. Why is OOP preferred over procedural programming for most of today's new programs? Here are some benefits of OOP:

- OOP programs have a more natural organization.

  Since people tend to think about real-world problems in terms of real-world objects, it's easier for people to understand a program that's organized around objects.

- OOP makes it easier to develop and maintain large programs.

  Although switching to OOP

programming typically makes a small program more complicated, it naturally partitions things so that the program grows gracefully and does not evolve into a giant mess. Since objects provide encapsulation, bugs (errors) and bug repairs tend to be localized.

The second bullet item needs some clarification. When an object's data can be modified only by using one of that object's methods, it's hard for a programmer to mess up an object's data accidentally. Returning again to the employee-salaries program example, assume the only way to change an employee object's salary is to use its adjustSalary method. Then, if there's a bug relating to an employee's salary, the programmer immediately knows where to look for the problem—in the adjustSalary method or in one of the calls to the adjustSalary method.

Classes

Having discussed objects, it’s now time to talk about an intimately related entity—a *class*. We’ll start with a broad definition of a class, and we’ll refine it later. Broadly speaking, a class is a description of all the objects it defines. As such, it is an *abstraction*—a concept apart from any particular instances. In Figure 6.2, note the three computers on a conveyor belt in a manufacturing plant. The three computers represent objects. The specifications document that hovers above the computers is a blueprint that describes the computers: it lists the computers’ components and describes the computers’ features. The computer-specification document represents a class. Each object is an instance of its class. Thus, for practical purposes, “object” and “instance” are synonyms.

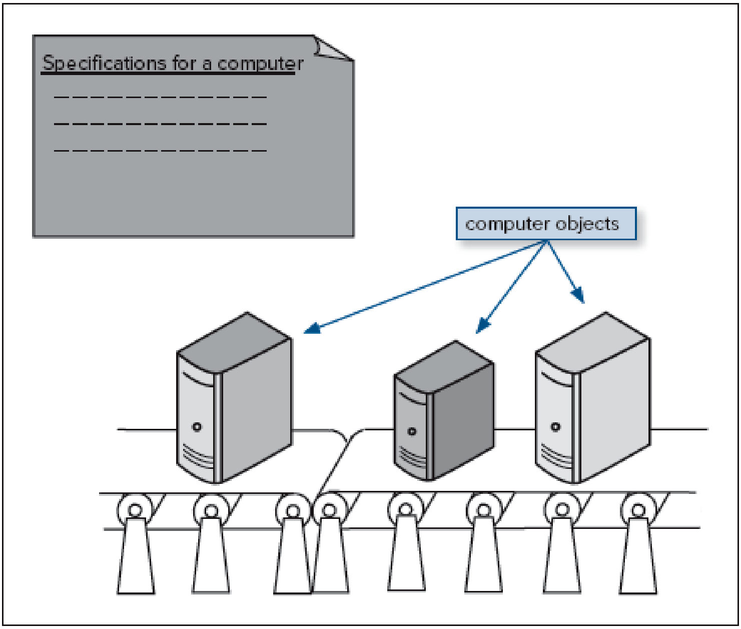

Specifications for a computer
computer objects

Figure 6.2 Conveyor belt portrayal of the class-objects relationship

One class can have any number of objects associated with it. A class can even have zero objects associated with it. This should make sense if you think about the computer-manufacturing example. Isn't it possible to have a blueprint for a computer, but not yet have any computers manufactured from that blueprint?

We'll now present a more complete description of a class. Above, we said that a class is a description for a set of objects. The description consists of:

a list of fields
+ a list of methods

In general programing language terms, a “field” is a storage device, which stores a particular type of information. In Java, a *field* is a variable by default, but if its declaration has a final modifier, it is a named constant. So, it may be either a variable or a constant.

Classes can define two types of fields—*static fields* and *instance fields*. And classes can define two types of methods—*static methods* and *instance methods.* Chapter 5 showed you how to use the Math class’s static methods, and you’ve been implementing the main method, a static method, since your first Java program. In Chapter 7, we’ll show you when it’s appropriate to define other static methods and define and use static fields. But it's easy to fall into the trap of defining and using static methods and static fields improperly. We want to keep you away from that trap until after you've developed good OOP habits. Therefore, we focus on instance fields and instance methods throughout this chapter and the next several chapters. But instead of using the generic term, field, we'll use the more specific terms, variable and constant.

A class's instance variables and instance constants specify the type of data that an object can store. For example, if you have a class for computer objects, and the Computer class contains a hardDiskSize instance variable, then each computer object stores a value for the size of the computer's hard disk. A class's instance methods specify the behavior that an object can exhibit. For example, if you have a class for computer objects, and the Computer class contains a printSpecifications instance method, then each computer object can print a specifications report (the specifications report shows the computer's hard disk size, CPU speed, cost, etc.).

The term "instance" in "instance variable" and "instance method" reinforces the fact that instance variables and instance methods are associated with a particular object instance. For example, each employee object would have its own value for a salary instance variable, which would be accessed through its adjustSalary instance method. That contrasts with static methods. Static methods are associated with an entire class. For example, the Math class contains the round static method, which is not associated with a particular instance of the Math class.

Type Inferencing

In Chapter 4 we described type inferencing in the context of variables that are local to a method (e.g., the main method) or a for loop—those variables are known as local variables. For local variables, type inferencing allows a programmer to substitute the reserved type name, var, for the type of an initializing local variable declaration. Note that type inferencing works only for local variables, and not for fields. Although it would be theoretically possible for the compiler to infer these types, Java designers have been reluctant to allow this because they want to avoid action-at-a-distance programming errors, where a programmer changes code at one place (where the variable is declared in this type inferencing scenario) and forgets to alter dependent code at other places. The danger of such errors increases as scope increases, so Java designers decided to limit the use of var to local variables.

Nevertheless, Java does employ variations of type inferencing in chaining operations, first seen in Section 4.8, and for the method parameters in *lambda expressions*, described later in Chapter 12 and utilized in Chapters 17, 18, and 19.

6.3 First OOP Class

Use UML to specify OOP.

In the next several sections, we put what you've learned into practice by implementing a complete OOP program. The program will contain a Mouse class, and it will simulate the growth of two Mouse objects (we're talking about rodents here, not computer pointing devices). As is customary with OOP programs, we start the implementation process by describing the solution pictorially with a UML class diagram. A UML class diagram is a diagrammatic technique for describing classes, objects, and the relationships between them. It is widely accepted in the software industry as a standard for modeling OOP designs. After describing our mouse-simulation solution with a UML class diagram, we will present the Mouse program's source code and walk you through it.

UML Class Diagram

See Figure 6.3. It contains an abbreviated UML class diagram for a Mouse class. A UML class diagram box is divided into three parts—class name at the top, *attributes* in the middle, and *operations* at the bottom. With Java programs, attributes equate to variables and operations equate to methods. Henceforth, we'll use the Java terms, *variables* and *methods,* rather than the formal UML terms, *attributes* and *operations*. Collectively, we refer to a class's variables and methods as the class's *members*. Let's now describe each Mouse member.

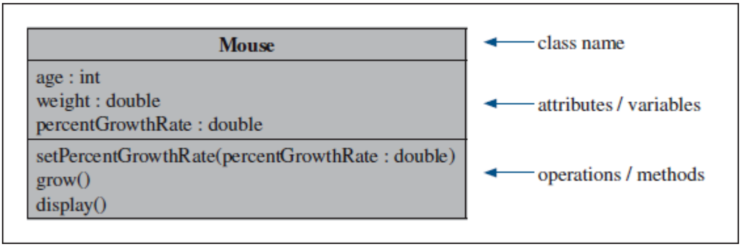

Mouse
age : int
weight : double
percentGrowthRate : double
setPercentGrowthRate(percentGrowthRate : double)
grow()
display()
class name
attributes / variables
operations / methods

Figure 6.3 Abbreviated UML class diagram for a Mouse class

The Mouse class has three instance variables—age, weight, and percentGrowthRate. The age instance variable keeps track of how old a Mouse object is, in days. The weight instance variable keeps track of a Mouse object's weight, in grams. The percentGrowthRate instance variable is the percentage of its current weight that gets added to its weight each day. If the percentGrowthRate is 10 percent and the mouse's current weight is 10 grams, then the mouse gains 1 gram by the next day.

The Mouse class has three instance methods—setPercentGrowthRate, grow, and display. The setPercentGrowthRate method assigns a specified value to the percentGrowthRate instance variable. The grow method simulates one day of weight gain for a mouse. The display method prints a mouse's age and weight.

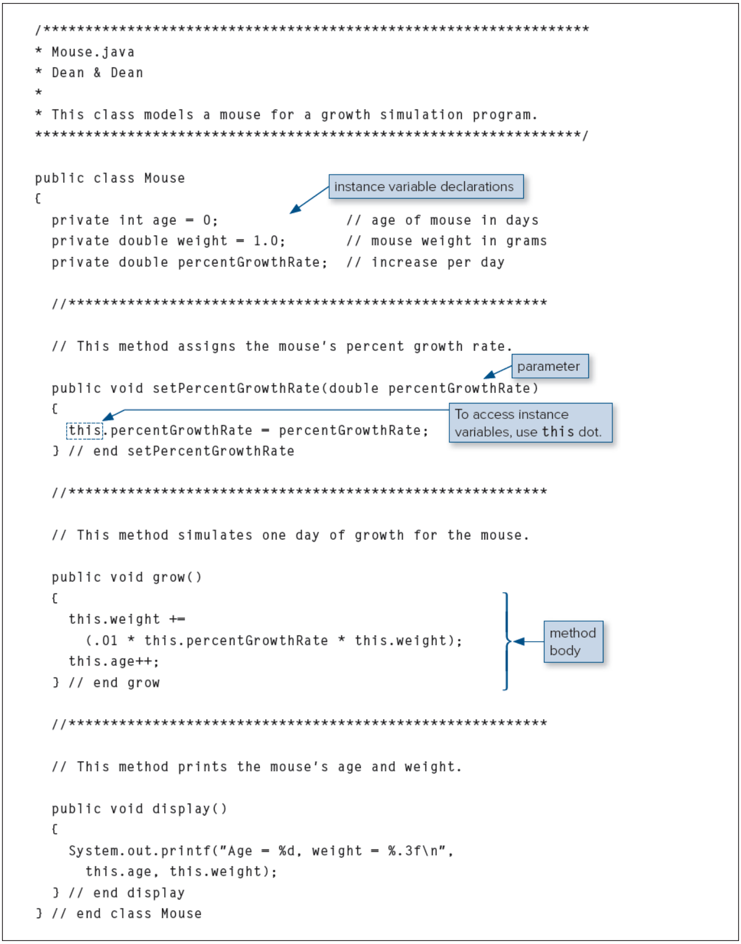
Referring to Figure 6.3, note how we specify variable types in a class diagram. The type appears at the right of the variable (e.g., age : int). That's opposite from Java declarations, where we write the type at the left of the variable (e.g., int age;)

Start documenting early.

Some programmers use UML class diagrams as a means to document programs after they've already been written. That's OK, but it's not how class diagrams were originally intended to be used. We encourage you to start drawing class diagrams as a first step in your solution implementation. The class diagram details provide an outline for your program. Depending on the complexity of the program and your affinity for pseudocode, you may want to code the methods directly with Java, or you may want to code the methods first with pseudocode as an intermediate step. For our Mouse example, the Mouse class's methods are straightforward, so we'll code them directly with Java. Let's now take a look at the Mouse class's Java source code.

Mouse Class Source Code

Figure 6.4 shows the Mouse class implemented with Java. Note the Mouse class's three instance variable declarations for age, weight, and percentGrowthRate. Instance variables must be declared outside all methods, and to make your code more self-documenting, you should declare them all at the beginning of the class definition. Instance variable declarations are very similar to variable declarations you've seen in the past: The variable's type goes at the left of the variable, and you can optionally assign an initial value to the variable. Do you remember what it's called when you assign a value to a variable as part of a declaration? That's called an *initialization*. Note the initializations for age and weight. We initialize age to 0 because newborn mice are zero days old. We initialize weight to 1 because newborn mice weigh approximately 1 gram.



```
/*******************************************************************
* Mouse.java
* Dean & Dean
*
* This class models a mouse for a growth simulation program.
*******************************************************************/

public class Mouse
{
  private int age = 0;                  // age of mouse in days
  private double weight = 1.0;          // mouse weight in grams
  private double percentGrowthRate;     // increase per day

  //************************************************************

  // This method assigns the mouse's percent growth rate.

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate

  //************************************************************

  // This method simulates one day of growth for the mouse.

  public void grow()
  {
    this.weight +=
      (.01 * this.percentGrowthRate * this.weight);
    this.age++;
  } // end grow

  //************************************************************

  // This method prints the mouse's age and weight.

  public void display()
  {
    System.out.printf("Age = %d, weight = %.3f\n",
      this.age, this.weight);
  } // end display
} // end class Mouse
```

instance variable declarations

parameter

To access instance variables, use `this` dot.

method body

Figure 6.4 Mouse class

The primary difference between instance variable declarations and variable declarations you've seen in the past is the private access modifier. If you declare a member to be private, then the member can be accessed only from within the member's class and not from the “outside world” (i.e., by code that’s outside of the class in which the member resides). Instance variables are almost always declared with the private access modifier because you almost always want an object’s data to be hidden. Making an instance variable private gives you control over how its value can be changed. For example, you could assure that a weight is never made negative. Constraining data access is what encapsulation is all about, and it’s one of the cornerstones of OOP.

In addition to the private access modifier, there’s also a public access modifier. Given the standard definitions of the words “public” and “private,” you can probably surmise that public members are easier to access than private members. If you declare a member to be public, then the member can be accessed from anywhere (from within the member’s class, and also from outside the member's class). You should declare a method to be public when you want it to be a portal through which the outside world accesses your objects' data. Go back and verify that all three methods in the Mouse class use the public access modifier. When you want a method to help perform a local task only, you should declare it to be private, but we'll delay that consideration until Chapter 8.

Look once again at the Mouse class's instance variable declarations. Note that we initialize age and weight to 0 and 1.0, respectively, but we don't initialize percentGrowthRate. That's because we're comfortable with age = 0 and weight = 1.0 for all newborn Mouse objects, but we're not comfortable with a predefined initial value for percentGrowthRate. Presumably, we'll want to use different percentGrowthRate values for different Mouse objects (mice in a doughnut-eating study might have higher percentGrowthRate values than mice in a cigarette-smoking study).

With no initialization for the percentGrowthRate instance variable, how can you set the growth rate for a Mouse object? You can have the Mouse object call the setPercentGrowthRate method with a growth rate value as an argument. For example, here's how a Mouse object can set its growth rate to 10 (percent):

```
setPercentGrowthRate(10);
```

As you may recall from Chapter 5, a method call's parenthetical values are referred to as *arguments*. Thus, in this example, 10 is an argument. The 10 gets passed into the percentGrowthRate variable in setPercentGrowthRate's heading. A method heading's parenthetical variables are referred to as *parameters*.

Thus, in the method heading in Figure 6.4, percentGrowthRate is a parameter. Within the setPercentGrowthRate *method body* (the code between the method's opening and closing braces), the percentGrowthRate parameter is assigned into the percentGrowthRate instance variable. Here's the relevant assignment statement:

```
this.percentGrowthRate = percentGrowthRate;
```

Note the "this dot" in this.percentGrowthRate. The this dot is how you tell the Java compiler that the variable you're referring to is an instance variable. Because the percentGrowthRate variable at the right does not have this dot, the Java compiler knows that that percentGrowthRate refers to the percentGrowthRate parameter, not the percentGrowthRate instance variable. In Figure 6.4's setPercentGrowthRate method, the instance variable and the parameter have the same name. That's a common practice. There’s no problem distinguishing between the two variables because the instance variable uses this dot and the parameter does not.

Now, take a look at the Mouse class’s display and grow methods. The display method is straightforward; it prints a mouse’s age and weight. The grow method simulates one day of weight gain for a mouse. The weight-gain formula adds a certain percentage of the current weight to the current weight. That means that the mouse will continue to grow every day of its life. That’s a simple, but not very accurate, portrayal of normal weight gain. We’ve intentionally kept the weight-gain formula simple in order to avoid getting bogged down in complicated math. In the final section of this chapter, we provide more realistic growth models.

Finally, take a look at the Mouse

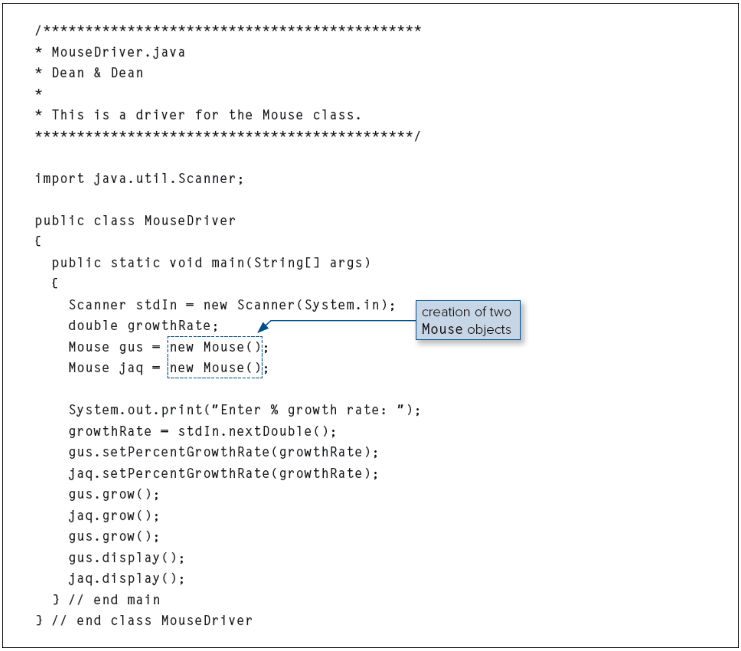
class's comments. Note the descriptions above each method. Proper style suggests that, above each method, you should have a blank line, a line of asterisks, a blank line, a description of the method, and another blank line. The blank lines and asterisks serve to separate the methods. The method descriptions allow someone who's reading your program to quickly get an idea of what's going on.

6.4 Driver Class

What Is a Driver?

Driver is a common computer term that applies to a piece of software that runs or "drives" something else. For example, a printer driver is a program that is in charge of running a printer. Likewise, a *driver class* is a class that is in charge of running another class.

In Figure 6.5, we present a MouseDriver class. We name the class MouseDriver because it is in charge of driving the Mouse class. We say that the MouseDriver class drives the Mouse class because it creates Mouse objects and then manipulates them. For example, note the gus = new Mouse() and the jaq = new Mouse() statements. That code creates Mouse objects gus and jaq.[1] In addition, note the gus.setPercentGrowthRate(growthRate) code. That code manipulates the gus object by updating gus's percentGrowthRate value.



```
/*********************************************
* MouseDriver.java
* Dean & Dean
*
* This is a driver for the Mouse class.
*********************************************/

import java.util.Scanner;

public class MouseDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double growthRate;
    Mouse gus = new Mouse();
    Mouse jaq = new Mouse();

    System.out.print("Enter % growth rate: ");
    growthRate = stdIn.nextDouble();
    gus.setPercentGrowthRate(growthRate);
    jaq.setPercentGrowthRate(growthRate);
    gus.grow();
    jaq.grow();
    gus.grow();
    gus.display();
    jaq.display();
  } // end main
} // end class MouseDriver
```

creation of two Mouse objects

Figure 6.5 MouseDriver class that drives the Mouse class in Figure 6.4

Normally, a driver class consists entirely of a main method and nothing else. The driver class, with its main method, is the starting point for the program. It calls upon the driven class to create objects and manipulate them. The driven class dutifully carries out the object creation and object manipulation requests. Normally, carrying out those tasks is the primary focus of the program, and their implementation requires the majority of the program's code. Thus, driven classes are typically (but not always) longer than driver classes.

Driver classes, such as the MouseDriver class, are in separate files from the classes that they drive. To make them accessible from the outside world, driver classes must be public. Each public class must be stored in a separate file whose name is the same as the class name, so the MouseDriver class must be stored in a file named MouseDriver.java. For MouseDriver's code to find the Mouse class, both classes should be in the same directory.2

Reference Variables

In the MouseDriver class, we create Mouse objects, and we refer to those Mouse objects using gus and jaq, where gus and jaq are *reference* *variables*. The value contained in a reference variable is a “reference” to an object (thus the name *reference variable*). More precisely, a reference variable holds the address of where an object is stored in memory. For a pictorial explanation, see Figure 6.6. In the figure, the little boxes immediately to the right of gus and jaq represent addresses. So gus’s little box holds the address of the first object.

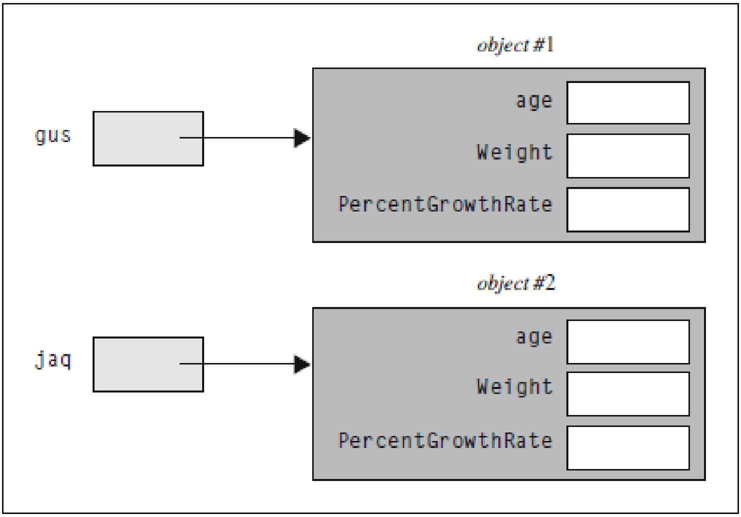

object #1
gus
age
Weight
PercentGrowthRate
object #2
jaq
age
Weight
PercentGrowthRate

Figure 6.6 Reference variables and objects for the Mouse program in Figures 6.4 and 6.5

The two reference variables on the left, gus and jaq, contain references that point to the two objects on the right.

Industry OOP Vernacular

Most Java programmers in industry don't use the term *reference variable*. Instead, they just use the term *object*. This blurs the distinction between reference variables and objects. For example, in the MouseDriver class in Figure 6.5, this statement initializes the gus reference variable:

```
Mouse gus = new Mouse();
```

Even though it's a reference variable, most industry Java programmers would refer to gus as an object. Despite the common practice of using the word "*object*" as a substitute for "*reference variable,*" it's important to know the difference—an object holds a group of data, and a reference variable holds the location where that group of data is stored in memory. Understanding the difference between an object and a reference variable will help you to understand the behavior of Java code.

Page 233

Declaring a Reference Variable

You must always declare a variable before you can use it. For example, in order to use an int variable named count, you must first declare count like this:

```
int count;
```

Likewise, in order to use a gus reference variable, you must first declare gus like this:

```
Mouse gus;
```

As you can see, the process for declaring reference variables mirrors the process for declaring primitive variables. The only difference is that instead of writing a primitive type on the left (e.g., int), for reference variables, you write a class name on the left (e.g., Mouse).

Instantiation and Assigning a Value to a Reference Variable

As you know, the point of a reference variable is to store a reference to an object. But before you can store a reference to an object, you have to have an object. So let's look at object creation.

To create an object, use the new operator. For example, to create a Mouse object, specify new Mouse(). The new operator should make sense when you realize that new Mouse() creates a new object. The formal term for creating an object is *instantiating* an object. So new Mouse() instantiates an object. The term *instantiate* is a verbalized form of the noun "instance." It is computer jargon for "make an instance of a class" or "create an object."

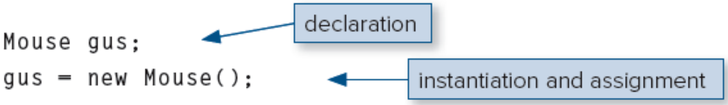
After instantiating an object, you'll

normally assign it to a reference variable. For example, to assign a Mouse object to the gus reference variable, do this:

```
gus = new Mouse();
```

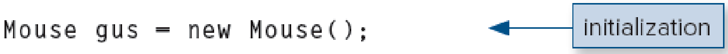
After the assignment, gus holds a reference to the newly created Mouse object.

Let's review. Here's how we declared a gus reference variable, instantiated a Mouse object, and assigned the object's address to gus:



```
Mouse gus;
gus = new Mouse();
```

Now here's how to do the same thing with only one statement:



```
Mouse gus = new Mouse();
```

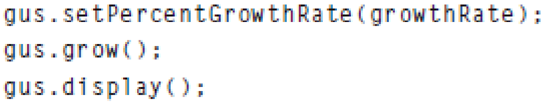
The above statement is what appears in Figure 6.5's MouseDriver class. It's an initialization. As mentioned previously, an initialization is when you declare a variable and assign it a value, all in one statement.

Calling a Method

After you instantiate an object and assign its reference to a reference variable, you can call/invoke an instance method using this syntax:

reference-variable.method-name(*comma-separated-arguments*);

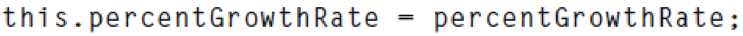
Here are three example instance method calls from the MouseDriver class:



```
gus.setPercentGrowthRate(growthRate);
gus.grow();
gus.display();
```

Note how the three method calls mimic the syntax template. The first method call has one argument and the next two method calls have zero arguments. If we had a method with two parameters, we'd call it with two arguments separated by a comma.

When a program calls a method, it passes control from the calling statement to the first executable statement in the called method. For example, when the MouseDriver’s main method calls the setPercentGrowthRate method with gus.setPercentGrowthRate(growthRate), control passes to this statement in the Mouse class’s setPercentGrowthRate method:



```
this.percentGrowthRate = percentGrowthRate;
```

Go back to Figure 6.4’s Mouse class and verify that the setPercentGrowthRate method contains the above statement.

After the last statement in any called method executes, control returns to the calling method at the point just after where the call was made. For a pictorial explanation, see Figure 6.7.

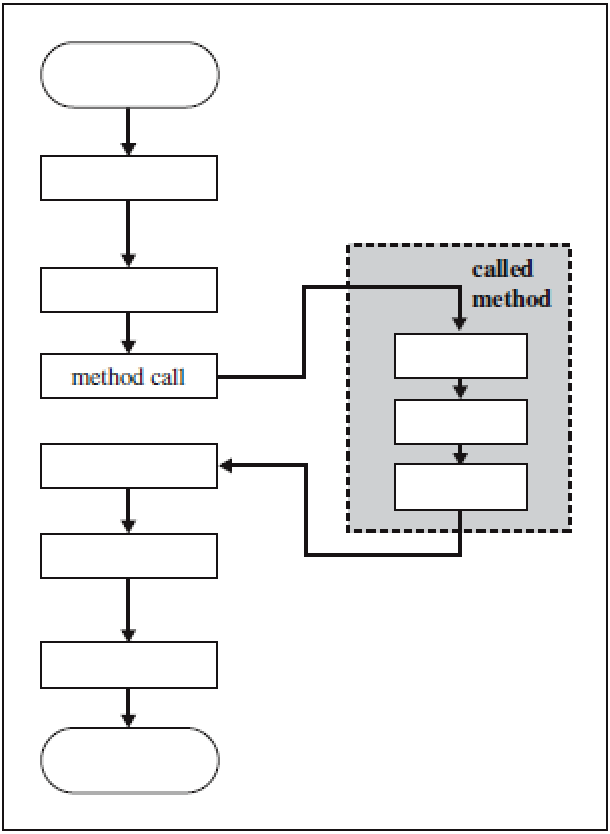

called
method
method call

Figure 6.7 Calling a method

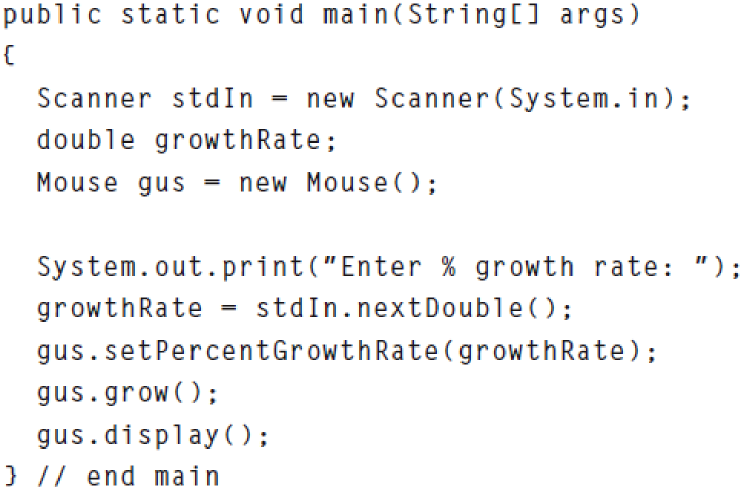
6.5 Calling Object, this Reference

Suppose you have two objects that are instances of the same class. For example, gus and jaq refer to two objects that are instances of the

Mouse class. And suppose you want the two objects to call the same instance method. For example, you want both gus and jaq to call setPercentGrowthRate. For each method call, the Java virtual machine (JVM) needs to know which object to update (if gus calls setPercentGrowthRate, then the JVM should update gus's percentGrowthRate; if jaq calls setPercentGrowthRate, then the JVM should update jaq's percentGrowthRate). This section describes how the JVM knows which object to update

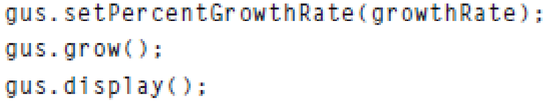
Calling Object

As mentioned in Chapter 5, whenever an instance method is called, it is associated with a calling object. You can identify the calling object by looking to the left of the dot in an instance method call statement. Can you identify the calling objects in the following main method?



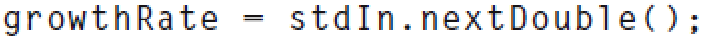
```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  double growthRate;
  Mouse gus = new Mouse();

  System.out.print("Enter % growth rate: ");
  growthRate = stdIn.nextDouble();
  gus.setPercentGrowthRate(growthRate);
  gus.grow();
  gus.display();
} // end main
```

The gus object is the calling object for these statements:



```
gus.setPercentGrowthRate(growthRate);
gus.grow();
gus.display();
```

Are there any other calling objects? Yes. The stdIn object is a calling object in this statement:



```
growthRate = stdIn.nextDouble();
```

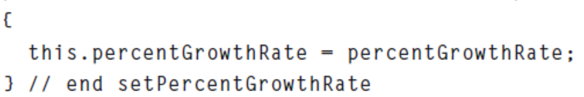
The this Reference

It's easy to identify the calling object when you're looking at a method call statement. But what if you're inside the called method—how can you tell what object called the method? For example, when you're looking at the definition of the Mouse class in Figure 6.4, can you identify the calling object that called its grow method? Here is that method again:

```
public void grow()
{
  this.weight +=
    (0.01 * this.percentGrowthRate * this.weight);
  this.age++;
} // end grow
```

The pronoun this (called the *this reference*) stands for the calling object, but it doesn't tell you which object that is. Thus, you can't tell what the calling object is just by looking at the method that was called. You must look at what called that method. If the statement that called grow was gus.grow(), then gus is the calling object. Alternately, if the statement that called grow was jaq.grow(), then jaq is the calling object. As you'll see when we do the upcoming trace, you must know which object, gus or jaq, is the current calling object so that you update the proper object. Within the above grow method, note this.weight and this.age. The this reference reminds you that weight and age are instance variables. Instance variables in which object? In the calling object!

The setPercentGrowthRate method in Figure 6.4 provides another example. Here is that method again:



```
public void setPercentGrowthRate(double percentGrowthRate)
{
  this.percentGrowthRate = percentGrowthRate;
} // end setPercentGrowthRate
```

The this reference tells you the variable on the left side of this method’s lone statement is an instance variable in the calling object. As indicated earlier, the this reference in this statement also helps the compiler and a human distinguish the variable on the left side from the variable on the right side. Before the advent of OOP, computer languages did not include this dot functionality. Then, the only way the compiler and a human could distinguish between variables in different places that referred to essentially the same thing was to give them similar but slightly different names.

The *ad hoc* (special case) nature of how old-time programmers devised slightly different names made programs confusing and increased programming errors. Java's this reference provides a standard way to make the distinction and show the relationship at the same time. You can use exactly the same name to show the relationship and then use this dot to make the distinction. So it is no longer necessary to use slightly different names for that purpose, and we recommend against that archaic practice.

To emphasize the meaning and utility of Java's this reference, we will use it with all examples of instance variables up to the end of the next chapter—even when it is not necessary to draw a distinction between an instance variable and a parameter. There is no performance penalty in using this dot, and it provides an immediate indicator to everyone that the variable is an instance variable. Thus, it helps to explain the program; that is, it provides useful self documentation.

6.6 Instance Variables

You’ve been exposed to instance variables for a while now. You know that an object stores its data in instance variables. You know that an instance method accesses its instance variables by prefacing them with the this reference (e.g., this.weight). In this section, we consider a few more instance variable details. Specifically, we consider default values and persistence.

Default Values for Instance Variables

As implied by the common definition of “default,” a variable’s *default value* is the variable’s value when there’s no explicitly assigned initial value; that is, when there’s no initialization. Different types of variables have different default values.

There are two integer types that we’ve covered so far—int and long. Integer-type instance variables are assigned 0 by default. But in the Mouse class, notice that we initialize the age instance variable to 0:

```
private int age = 0; // age of mouse in days
```

Why bother with the explicit initialization? Wouldn’t age be assigned 0 by default even if “= 0” was omitted? Yes, the program would work the same either way. But it’s poor practice to depend on hidden default values. By explicitly assigning values to variables, we show our intent. That’s a form of self-documenting code.

There are two floating-point types—float and double. Floating-point-type instance variables are assigned 0.0 by default. The Mouse class declares two floating-point instance variables—weight and percentGrowthRate:

```
private double weight = 1.0;      // weight of mouse in grams
private double percentGrowthRate; // % weight increase per day
```

In this case, we initialize the weight instance variable to 1.0, so the default value doesn’t come into play. We do not initialize the percentGrowthRate value, so percentGrowthRate is initialized to 0.0 by default. Didn’t we just say that it’s poor practice to depend on hidden default values? Yes, but in this case, we’re not depending on the default value. In the MouseDriver class, we overlay the percentGrowthRate default value with a custom value by calling setPercentGrowthRate like this:

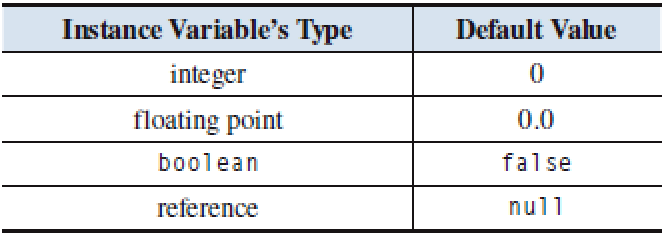
```
gus.setPercentGrowthRate(growthRate);
```

By default, boolean instance variables are assigned false. For example, if you added a boolean instance variable named vaccinated to the Mouse class, vaccinated would be assigned false by default.

Reference-type instance variables are assigned null by default. For example, if you added a String instance variable named breed to the Mouse class, breed would be assigned null by default. Normally, a reference variable holds the address of an object and that address points to an object. The Java designers added null to the language as a way to indicate that a reference variable points to nothing. So the default for a reference-type instance variable is to point to nothing.

Here’s a summary of default values

for instance variables:



Instance Variable's Type	Default Value
integer	0
floating point	0.0
`boolean`	`false`
reference	`null`

Instance Variable Persistence

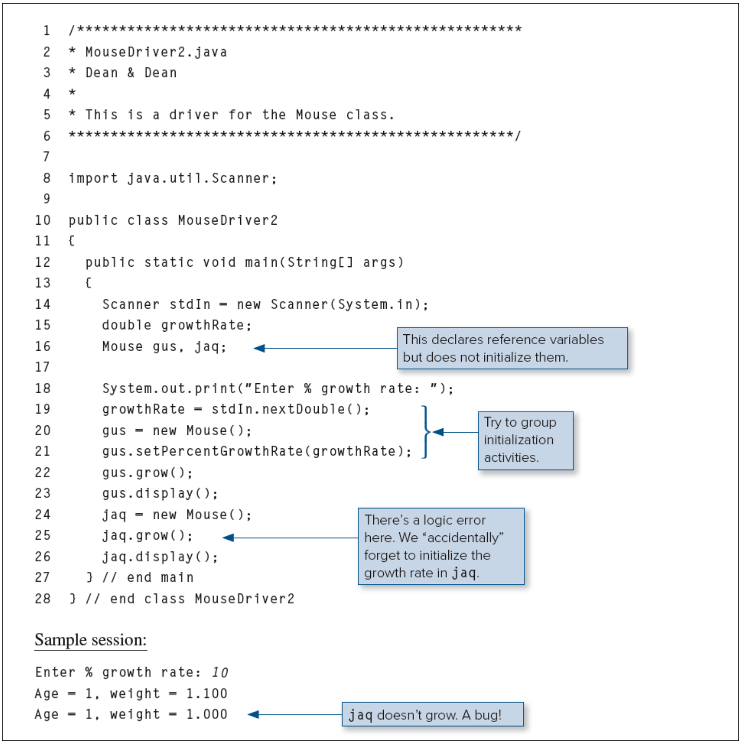
Now consider variable *persistence*. Persistence refers to how long a variable's value survives before it's wiped out. Instance variables persist for the duration of a particular object. Thus, if an object makes two method calls, the second called method does not reset the calling object's instance variables to their initialized values. Instead, the object's instance variables retain their values from one method call to the next. For example, in the MouseDriver class, gus calls grow twice. In the first call to grow, gus's age increments from 0 to 1. In the second call to grow, gus's age starts out as 1 and increments to 2. gus's age retains its value from one grow call to the next because age is an instance variable.

6.7 Tracing an OOP Program

To reinforce what you've learned so far in this chapter, we'll trace the Mouse program. Remember the tracing procedure we used in prior chapters? It worked fine for programs with only one method—the main method. But for OOP programs with multiple classes and multiple methods, you'll need to keep track of which class and which method you're in and which object called that method. In addition, you'll need to keep track of parameters and instance variables. This requires a more elaborate trace table.

Use trace to find the cause of a problem.

In tracing the Mouse program, we'll use a slightly different driver, the MouseDriver2 class, shown in Figure 6.8. In MouseDriver2, we delay the instantiation of the individual mice and assign their growth rates (by calling setPercentGrowthRate) immediately after each instantiation. This is better style, because it more closely associates each object's instantiation with its growth rate assignment. However, in changing the driver, we "accidentally" forget to call setPercentGrowthRate for jaq, the second mouse. You can see the effect of this logic error in the output—jaq doesn't grow (after the first day, jaq still weighs 1 gram). But let's pretend that you don't know why this error occurs and use the trace to help find its cause. Remember—tracing is an effective tool when you need help debugging a program.



```
/*********************************************************
* MouseDriver2.java
* Dean & Dean
*
* This is a driver for the Mouse class.
*********************************************************/

import java.util.Scanner;

public class MouseDriver2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double growthRate;
    Mouse gus, jaq;

    System.out.print("Enter % growth rate: ");
    growthRate = stdIn.nextDouble();
    gus = new Mouse();
    gus.setPercentGrowthRate(growthRate);
    gus.grow();
    gus.display();
    jaq = new Mouse();
    jaq.grow();
    jaq.display();
  } // end main
} // end class MouseDriver2
```

This declares reference variables but does not initialize them.

Try to group initialization activities.

There's a logic error here. We "accidentally" forget to initialize the growth rate in `jaq`.

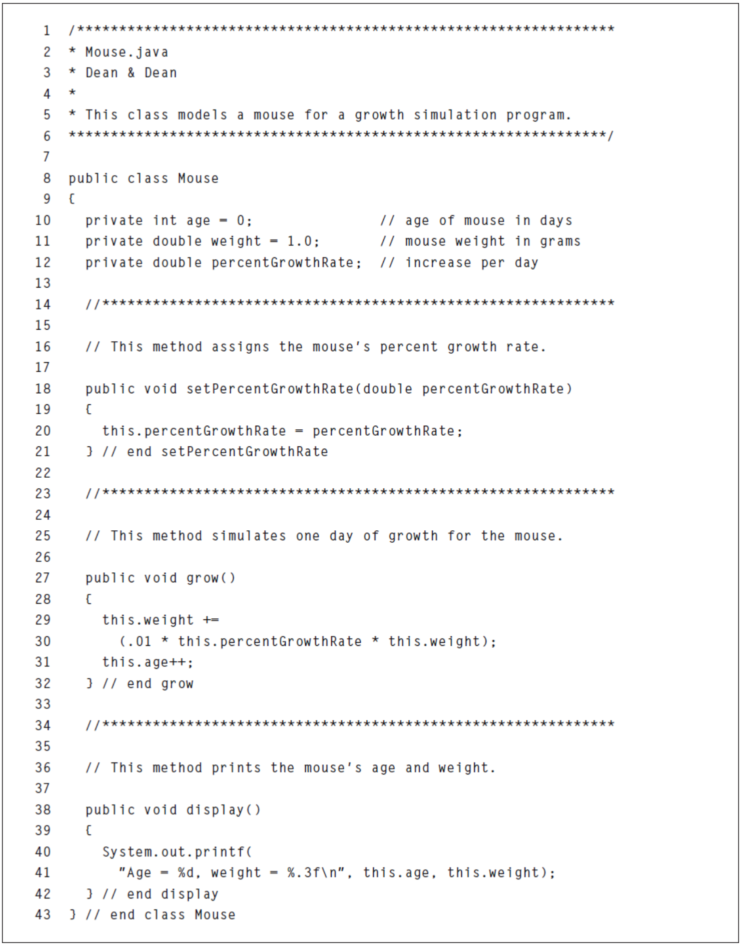
Sample session:

```
Enter % growth rate: 10
Age = 1, weight = 1.100
Age = 1, weight = 1.000
```

`jaq` doesn't grow. A bug!

Figure 6.8 MouseDriver2 class that drives the Mouse class in Figure 6.9

To perform the trace, in addition to the driver, you'll also need the code for the driven class. For your convenience, we repeat the original driven Mouse class in Figure 6.9.



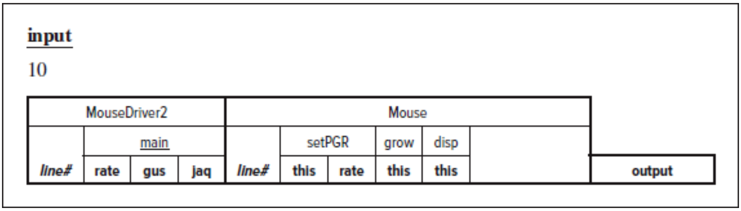
```java
/*********************************************************************
* Mouse.java
* Dean & Dean
*
* This class models a mouse for a growth simulation program.
*********************************************************************/

public class Mouse
{
  private int age = 0;                 // age of mouse in days
  private double weight = 1.0;         // mouse weight in grams
  private double percentGrowthRate;    // increase per day

  //*******************************************************************

  // This method assigns the mouse's percent growth rate.

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate

  //*******************************************************************

  // This method simulates one day of growth for the mouse.

  public void grow()
  {
    this.weight +=
      (.01 * this.percentGrowthRate * this.weight);
    this.age++;
  } // end grow

  //*******************************************************************

  // This method prints the mouse's age and weight.

  public void display()
  {
    System.out.printf(
      "Age = %d, weight = %.3f\n", this.age, this.weight);
  } // end display
} // end class Mouse
```

Figure 6.9 Mouse class, repeated from Figure 6.4

Page 240

Trace Setup

Figure 6.10 shows the setup. As with the traces in the previous chapters, the input goes in the top-left corner. Unlike the traces in the previous chapters, the headings under the input now require more than one line. The first line of headings shows the class names—MouseDriver2 and Mouse. Under each class name heading is a heading for each of the class's methods. In the trace setup, find the setPercentGrowthRate, grow, and display method headings (to save space, we abbreviated setPercentGrowthRate and display to setPGR and disp, respectively). And under each method-name heading, there's a heading for each of the method's local variables and parameters.



input

10

MouseDriver2				Mouse						
	main				setPGR		grow	disp		
line#	rate	gus	jaq	line#	this	rate	this	this		output

Figure 6.10 Trace setup for the Mouse program

We'll discuss *local variables* in detail later, but for now, just realize that growthRate (abbreviated to rate in the trace setup), gus, and jaq are considered to be local variables because they're declared and used "locally" within one particular method, the main method. That's different from the age, weight, and percentGrowthRate instance variables, which are declared outside of all methods, at the top of the class. Note that stdIn is another local variable within main, but there's no need to trace it because it's instantiated from an application programming interface (API) class, Scanner. There's no need to trace API classes because they've already been traced and tested thoroughly by the developers of the Java language. You can assume that they work properly.

Now let's examine the trace setup's parameters. The setPercentGrowthRate method has two parameters—

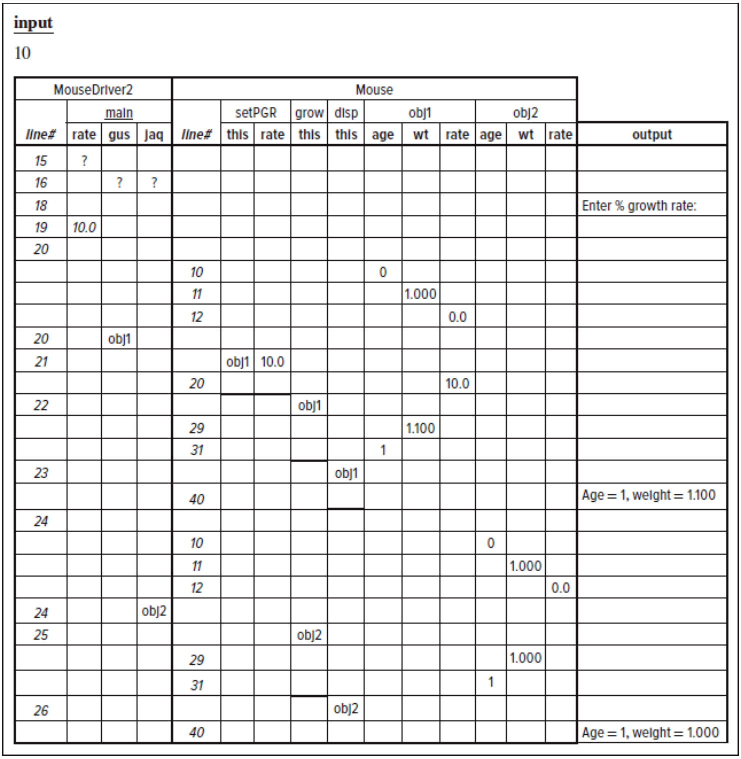
percentGrowthRate, abbreviated to rate in the trace setup, and the this reference, an implicit parameter. As you may recall, the this reference points to the calling object. For the setPercentGrowthRate, grow, and display methods, we include a column for this so that the trace can keep track of which object called the method.

Note the vacant area under the Mouse heading. We'll fill in more headings there as we execute the trace.

Trace Execution

Using Figure 6.10's trace setup as a starting point, we'll walk you through the key sections of the trace shown in Figure 6.11. We'll focus on the OOP parts of the trace since those are the parts that are new to you. When starting a method, under the method's local variable headings, write initial values for each of the local variables. Use a question mark for local variables that are uninitialized. In the first three lines of Figure 6.11's trace, note the ?'s for the uninitialized growthRate (abbreviated to rate), gus, and jaq local variables.

When an object is instantiated, under the object's class-name heading, provide a column heading named "obj#", where # is a unique number. Under the obj# heading, provide an underlined column heading for each of the object's instance variables. Under the instance variable headings, write initial values for each of the instance variables. In Figure 6.11's trace, note the obj1 and obj2 column headings and their age, weight, and percentGrowthRate (abbreviated to rate) subheadings. Also note the initial values for the age, weight, and percentGrowthRate instance variables.



input

10

MouseDriver2				Mouse											
	main				setPGR		grow	disp	obj1			obj2			
line#	rate	gus	jaq	line#	this	rate	this	this	age	wt	rate	age	wt	rate	output
15	?														
16		?	?												
18															Enter % growth rate:
19	10.0														
20															
				10					0						
				11						1.000					
				12							0.0				
20		obj1													
21					obj1	10.0									
				20							10.0				
22							obj1								
				29						1.100					
				31					1						
23								obj1							
				40											Age = 1, weight = 1.100
24															
				10								0			
				11									1.000		
				12										0.0	
24			obj2												
25							obj2								
				29									1.000		
				31								1			
26								obj2							
				40											Age = 1, weight = 1.000

Figure 6.11 Completed trace for the Mouse program

Page 241

When there's an assignment into a reference variable, write obj# under the reference variable's column heading, where obj# matches up with the associated obj# in the object portion of the trace. For example, in Figure 6.11's trace, we created obj1 while tracing the gus = new Mouse();

statement. Subsequently, we put obj1 under the gus column heading.

When there's a method call, under the called method's this column heading, write the calling object's obj#. In Figure 6.11's trace, note obj1 under setPercentGrowthRate's this heading. If the method call contains an argument, write the argument's value under the called method's associated parameter. In the trace, note the passed-in 10 under the setPercentGrowthRate's percentGrowthRate heading. Inside the method, if there's a this reference, find the obj# under the method's this column heading. Then go to the found obj#'s heading and read or update the obj#'s value accordingly. In Figure 6.9's Mouse class, note this.percentGrowthRate in the setPercentGrowthRate method body. In the trace, note that setPercentGrowthRate's this reference refers to obj1, so obj1's percentGrowthRate is updated accordingly.

When you finish tracing a method, draw a horizontal line under the method's variable values to indicate the end of the method trace and to signify that the values in the method's local variables are wiped out. For example, in the trace, the heavy horizontal line in Mouse line #20 under setPGR indicates the end of the setPercentGrowthRate method, and it signifies that percentGrowthRate's value is wiped out.

Practice.

Now that we've walked you through the new techniques for tracing an OOP program, we encourage you to go back to the trace setup in Figure 6.10 and do the entire trace on your own. Pay particular attention to what happens when gus and jaq call the grow method. Verify that gus's weight increases (as it should) and jaq's weight fails to increase (a bug). When you're done with the trace, compare your answer to Figure 6.11.

Paper trace emulates IDE debugger.

Experience with the long-form tracing used in this book will make it easier for you to understand what an automated debugger in an *Integrated Development Environment* (IDE) is telling you. As you step through a program that's running in debug mode under the control of an IDE debugger, when you get to a method call, you have two choices. You can "step into" and go through all the statements in the called method, like we do in Figure 6.11, or you can "step over" and just see what happens after the method returns. In a typical debugging activity, you will use a combination of stepping over and stepping in. For the example problem we have been considering, the sample session in Figure 6.8 tells you that the simulation is OK for the first object. The problem is with the second object. So, the appropriate thing to do is step over the method calls down through line 23 in the MouseDriver2 class. Then, starting at line 24 in the MouseDriver2 class, step into the methods calls to zero in on what caused the problem.

6.8 UML Class Diagrams

The Mouse class's grow method is not very flexible—it forces the driver to call the grow method separately for each day or to provide a for loop for each multiple-day simulation. It isn't good style to include such things in a driver. It's better to include multiple-day functionality within the driven class. In this section, we do just that. We present a revised mouse class with a grow method that handles any number of days, not just one day.

Organize.

To specify a second-generation mouse class (Mouse2) and an associated driver class (Mouse2Driver), let's create another UML class diagram. The diagram we presented in Figure 6.3 was a pared-down UML class diagram. It did not include all the standard features. This time, in Figure 6.12, we present a UML class diagram that includes all the standard features, plus an extra feature.

Figure 6.12's class diagram includes class diagram boxes for both classes—one diagram for the Mouse2Driver class and another diagram for the Mouse2 class. The Mouse2 class has the same three instance variables as the original Mouse class—age, weight, and percentGrowthRate. It also has the same setPercentGrowthRate method.

But the getAge and getWeight methods are new, and the grow method is improved. The getAge method retrieves a mouse's age. Remember that the age variable is private, so the only way for the outside world to read a mouse object's age is to use a public method—the getAge method. The getWeight method retrieves a mouse's weight. The grow method simulates a mouse's growth for a specified number of days. Note the days parameter. The number of days is passed into the days parameter, and that's how the method knows how many days to simulate.

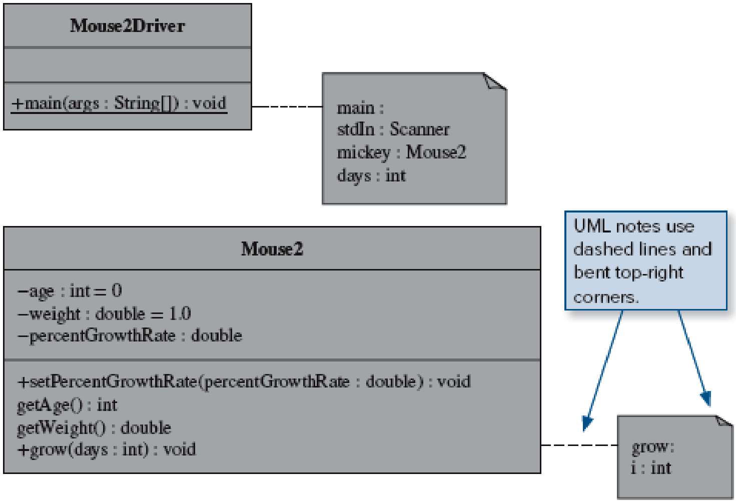

Mouse2Driver
+main(args : String[]) : void
main :
stdIn : Scanner
mickey : Mouse2
days : int
Mouse2
−age : int = 0
−weight : double = 1.0
−percentGrowthRate : double
+setPercentGrowthRate(percentGrowthRate : double) : void
getAge() : int
getWeight() : double
+grow(days : int) : void
UML notes use dashed lines and bent top-right corners.
grow:
i : int

Figure 6.12 UML class diagram for a second-generation Mouse program

Here are some of the standard UML class diagram features not found in Figure 6.3 that do appear in Figure 6.12:

- To specify member accessibility, prefix all member specifications with a “-” for private access or a “+” for public access. The instance variables have “-” prefixes, since we want them to be private, and the methods have “+” prefixes, since we want them to be public.

- To specify initialization, append “= *value*” to each variable declaration that includes initialization. For example, note the “= 0” after the age instance variable’s specification.
- Underline the main method in the MouseDriver class diagram box, since the main method is declared with the static modifier. UML standards suggest that you underline all methods and variables that are declared with the static modifier. As you learned in Chapter 5, the static modifier indicates a class member. You’ll learn more about class members in Chapter 7.
- Include a “: *type*” suffix with each method. This specifies the type of value that the method returns. All the methods in the Mouse class in Figure 6.4 returned void (nothing), but in Chapter 5,

you saw many Java API static methods with return types like int and double, and we'll discuss implementation of such methods later in this chapter.

Figure 6.12 also includes an extra UML class diagram feature. It has *notes* for two of its methods—the main and grow methods. The notes are depicted by the rectangles with the bent top-right corners. Why bent corners? They are supposed to give the impression of a piece of paper with its corner folded, an indication of a hardcopy "note." Including a note in a UML class diagram is purely optional. Usually we won't use them, but this time, we did because we wanted to show how you can include local variables in a UML class diagram.

6.9 Local Variables

A *local variable* is a variable that's declared and used "locally" inside a method. That's different from an instance variable, which is declared at the top of a class, outside all methods. As you perhaps now realize, all the variables we defined in chapters prior to this chapter were local variables. They were all declared within main methods, so they were all local variables within the main method. We didn't bother to explain the term "local variable" in much detail until now because there were no other methods besides main, and the idea of a variable being local to main wouldn't have made much sense. But the OOP context makes the concept of a local variable more meaningful.

Scope

A local variable has *local scope*—it can be used only from the point at which the variable is declared to the end of the variable’s block. A variable’s *block* is established by the closest pair of braces that enclose the variable’s declaration. Most of the time, you should declare a method’s local variables at the top of the method’s body. The scope of such variables is then the entire body of the method.

Index variables in a for loop are local variables, but they are special. Their scope rule is slightly different from what is described above. As you know from Chapter 4, you should normally declare a for loop’s index variable within the for loop’s header. The scope of such a variable is the for loop’s header plus the for loop’s body.

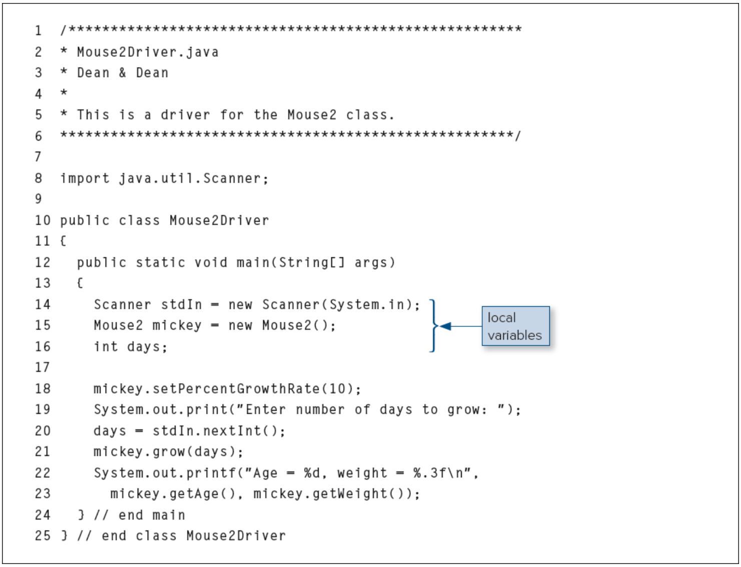
Method parameters are usually not considered to be local variables because they are initialized with the values of the corresponding method-call arguments. However, they are very similar to local variables in that they are declared and used “locally” inside a method. As with local variables, the scope of a method's parameters is limited to within the body of that method.

Let's round out the discussion of scope by comparing local scope to the scope used by instance variables. While variables with local scope can be accessed only within one particular method, instance variables can be accessed from within any instance method in the instance variable's class. Furthermore, if an instance variable is declared with the public access modifier, it can be accessed from outside of the instance variable's class (with the help of an instantiated object from the instance variable's class).

Mouse2Driver Class

To illustrate local variable principles, we present the Mouse2 program in Figures 6.13 and 6.14. The code includes line numbers to facilitate tracing in an end-of-chapter exercise. The main method in the Mouse2Driver class has three local variables—stdIn, mickey, and days. These appear in the UML class diagram note at the top of Figure 6.12, and they also appear as declarations in the main method in Figure 6.13.

Let's examine Figure 6.13's Mouse2Driver class. In the call to setPercentGrowthRate, note that we pass in a constant, 10, instead of a variable. Normally, you'll use variables for your arguments, but this example shows that it's also legal to use constants. After setting the percent growth rate, we prompt the user for the number of days of simulated growth, and then we pass the days value into the grow method. Then we print mickey's age and weight by embedding getAge and getWeight method calls within a printf statement.



```
/*******************************************************
* Mouse2Driver.java
* Dean & Dean
*
* This is a driver for the Mouse2 class.
*******************************************************/

import java.util.Scanner;

public class Mouse2Driver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Mouse2 mickey = new Mouse2();
    int days;

    mickey.setPercentGrowthRate(10);
    System.out.print("Enter number of days to grow: ");
    days = stdIn.nextInt();
    mickey.grow(days);
    System.out.printf("Age = %d, weight = %.3f\n",
      mickey.getAge(), mickey.getWeight());
  } // end main
} // end class Mouse2Driver
```

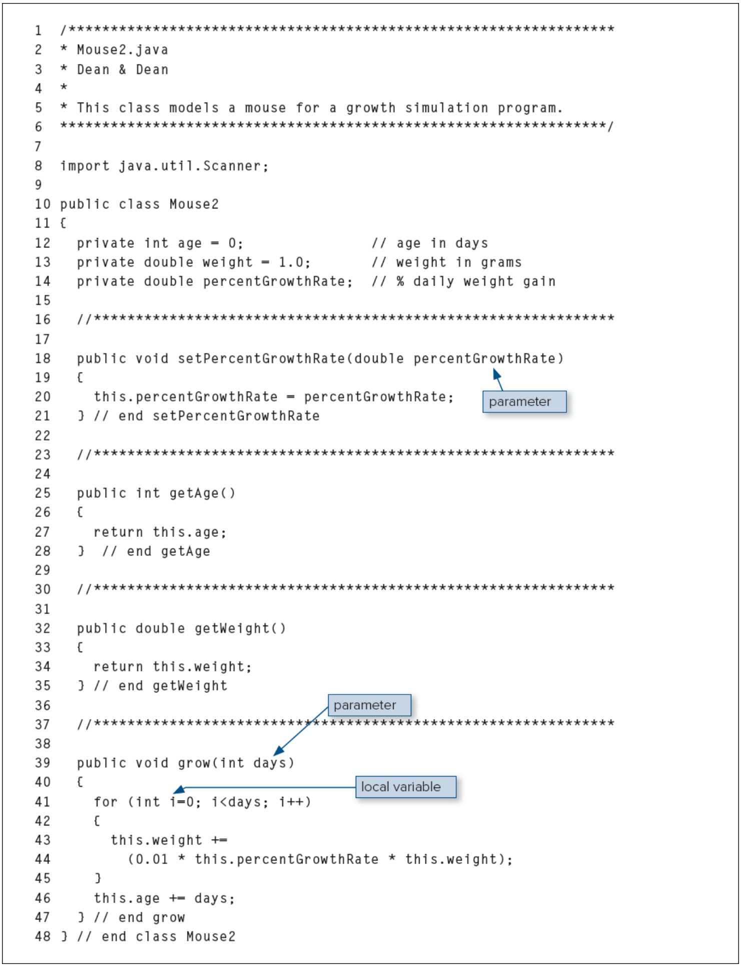
Figure 6.13 Mouse2Driver class that drives the Mouse2 class in Figure 6.14

Page 245

Mouse2 Class

Now look at the Mouse2 class in Figure 6.14. Are there any local variables there? The age, weight, and percentGrowthRate variables are instance variables, not local variables, because they're declared outside of all the methods, at the top of the class. Inside the grow method, we highlight this fact by prefixing each of these instance variables with a this reference. The grow method also includes a local variable—the i in the for loop. Since i is declared within the for loop header, its scope is limited to the for loop block. So you can read and update i only within the for loop. If you try to access i outside the for loop, you'll get a compilation error. This grow method is similar to the previous Mouse program's grow method, but this time we use a for loop to simulate multiple days of growth rather than just one day. The days parameter determines how many times the loop will repeat.

Previously, we described the default values for instance variables. Now, we'll describe the default values for local variables. Local variables contain *garbage* by default. Garbage means that the variable's value is unknown—it's whatever just happens to be in memory at the time that the variable is created. If a program attempts to access a variable that contains garbage, the compiler generates a compilation error. For example, what would happen if the =0 initialization were removed from the for loop header in the grow method in Figure 6.14? In other words, suppose that for loop was replaced by this:



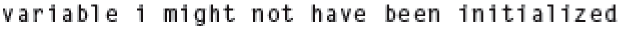
```
/*******************************************************************
* Mouse2.java
* Dean & Dean
*
* This class models a mouse for a growth simulation program.
*******************************************************************/

import java.util.Scanner;

public class Mouse2
{
  private int age = 0;                  // age in days
  private double weight = 1.0;          // weight in grams
  private double percentGrowthRate;    // % daily weight gain

  //*****************************************************************

  public void setPercentGrowthRate(double percentGrowthRate)
  {
    this.percentGrowthRate = percentGrowthRate;
  } // end setPercentGrowthRate

  //*****************************************************************

  public int getAge()
  {
    return this.age;
  }  // end getAge

  //*****************************************************************

  public double getWeight()
  {
    return this.weight;
  } // end getWeight

  //*****************************************************************

  public void grow(int days)
  {
    for (int i=0; i<days; i++)
    {
      this.weight +=
        (0.01 * this.percentGrowthRate * this.weight);
    }
    this.age += days;
  } // end grow
} // end class Mouse2
```

Figure 6.14 Mouse2 class

```
for (int i; i<days; i++)
{
  this.weight +=
    (0.01 * this.percentGrowthRate * this.weight);
}
```

Because i is no longer assigned zero, i contains garbage when the i<days

condition is tested. If you tried to compile code with a statement like this, it wouldn't compile, and the compiler would report: ⚠



```
variable i might not have been initialized
```

Local Variable Persistence

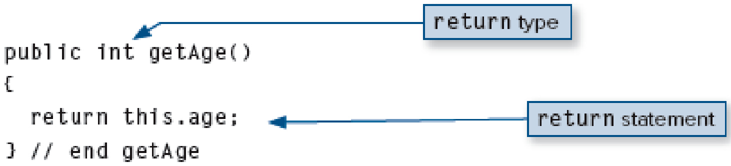
OK, let's say you do initialize a local variable. How long will it *persist*? A local variable (or parameter) persists only within its scope and only for the current duration of the method in which it is defined. The next time the method is called, the local variable's value resets to garbage or the value given to it by initialization. The horizontal line drawn in a trace after a method terminates reminds you that method termination converts all the method's local variables into garbage.

6.10 The return Statement

If you look back at our original Mouse class in Figures 6.4 and 6.10, you'll notice that every method heading has a void modifier located at the left of the method name. That means the method does not return any value, and we say "the method has a void return type" or more simply "it's a void method." But recall from Chapter 5 that many of the Java API methods return some kind of value, and in each case, the type of value returned is indicated by an appropriate return type in the method heading located at the left of the method name.

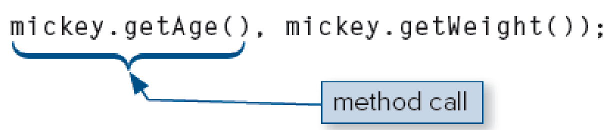
Returning a Value

If you look at the Mouse2 class in Figure 6.14, you'll see that two of the methods have a return type that is different from void. Here is one of those methods:



```
public int getAge()
{
  return this.age;
} // end getAge
```

The return statement in this method allows you to pass a value from the method back to where the method was called. In this case, the getAge method returns age to Mouse2Driver's printf statement in Figure 6.13. Here is that statement again:



```
System.out.printf("Age = %d, weight = %.3f\n",
  mickey.getAge(), mickey.getWeight());
```

In effect, the JVM "assigns" the return value (this.age) to the method call (mickey.getAge()). To perform a mental trace, imagine that the method call is overlaid by the returned value. So if Mickey's age is 2, then 2 is returned, and you can replace the getAge method call by the value 2.

Whenever a method heading's type is different from void, that method must return a value by means of a return statement, and the type of that value must match the type specified in the method heading. For example, the getAge method heading specifies an int return type. The return statement within the getAge method returns this.age. In Figure 6.14, the age instance variable was declared to be an int, and that matches getAge's int return type, so all is well. It's OK to have an expression following the word return; you aren't limited to just having a simple variable. But the expression must evaluate to the method's return type. For example, would it be legal to use this?

```
return this.age + 1;
```

Yes, because this.age + 1 evaluates to an int type, and that matches getAge's return type.

When a method includes conditional branching (with an if statement or a switch construct), it's possible to return from more than one place in the method. In such cases, all returns must match the type specified in the method heading.

Empty return Statement

For methods with a void return type, it's legal to have an *empty* return statement. The empty return statement looks like this:

```
return;
```

The empty return statement does what you'd expect. It terminates the current method and causes control to be passed back to the calling module at the point that immediately follows the method call. Here's a variation of our previous grow method that uses an empty return statement:

```
public void grow(int days)
{
  int endAge = this.age + days;

  while (this.age < endAge)
  {
    if (this.age >= 100)
    {
      return;
    }
    this.weight +=
      .01 * this.percentGrowthRate * this.weight;
    this.age++;
  } // end while
} // end grow
```

empty `return` statement

In this variation of the grow method, we cut off the aging process at 100 days—after "adolescence"—by checking age inside the loop and returning when age is greater than or equal to 100. Notice the empty return statement. Because nothing is returned, the method heading must specify void for its return type.

It would be illegal to have an empty return statement and a non-empty return statement in the same method. Why? Empty and non-empty return statements have different return types (void for an empty return statement and some other type for a non-empty return statement). There is no way to specify a type in the heading that simultaneously matches two different return types.

The empty return statement is a helpful statement in that it provides an easy way to exit quickly from a method. However, it does not provide unique functionality. Code that uses an empty return statement can always be replaced by code that is devoid of return statements. For example, here's a return-less version of the previous grow method:

```
public void grow(int days)
{
  int endAge = this.age + days;

  if (endAge > 100)
  {
    endAge = 100;
  }
  while (this.age < endAge)
  {
    this.weight +=
      .01 * this.percentGrowthRate * this.weight;
    this.age++;
  }  // end while
} // end grow
```

return Statement Within a Loop

Programmers in industry often are asked to maintain (fix and improve) other people's code. In doing that, they often find themselves having to examine the loops and, more specifically, the loop termination conditions in the program they're working on. Therefore, it's important that loop termination conditions are clear. Normally, loop termination conditions appear in the standard loop-condition section. For while loops, that's at the top, for do loops, that's at the bottom, and for for loops, that's in the header's second component. However, a return statement inside a loop results in a loop termination condition that's not in a standard location. For example, in the first grow method on the previous page, the return statement is inside an if statement and the loop termination condition is consequently "hidden" in the if statement's condition.

In the interest of maintainability, you should use restraint when considering the use of a return statement inside a loop. Based on the context, if inserting return statements inside a loop improves clarity, then feel free to insert. However, if it simply makes the coding chores easier and it does not add clarity, then don't insert. So which grow implementation is better—the empty return version or the return-less version? In general, we prefer the return-less version for maintainability reasons. However, because the code in both of our adolescent grow methods is so simple, it doesn't make much difference here.

6.11 Argument Passing

In the previous section, you saw that when a method finishes, the JVM effectively assigns the return value to the method call. This section describes a similar transfer in the other direction. When a method is called, the JVM effectively assigns the value of each argument in the calling statement to the corresponding parameter in the called method.

Example

Let's examine argument passing by looking at an example—another version of our Mouse program called Mouse3. Here is the code for this new version's driver:

```
public class Mouse3Driver
{
  public static void main(String[] args)
  {
    Mouse3 minnie = new Mouse3();
    int days = 365;

    minnie.grow(days);
    System.out.println("# of days aged = " + days);
  } // end main
} // end class Mouse3Driver
```

The JVM makes a copy of `days`'s value and passes it to the `grow` method.

The Mouse3Driver class calls the grow method with an argument called days, whose value happens to be 365. Then it assigns this value (365) to the parameter called days in the grow method. The following code shows what happens to the days parameter within the grow method:

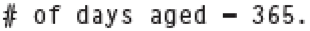
```
public class Mouse3
{
  private int age = 0;                         // age in days
  private double weight = 1.0;                 // weight in grams
  private double percentGrowthRate = 10;       // % daily weight gain
  public void grow(int days)
  {
    this.age += days;
    while (days > 0)
    {
      this.weight +=
        .01 * this.percentGrowthRate * this.weight;
      days--;
    }
  } // end grow
} // end class Mouse3
```

The JVM assigns the passed-in value to the `days` parameter.

The `days` parameter decrements down to 0.

Within a method, parameters are

treated like local variables. The only difference is that a local variable is initialized inside the method, whereas a parameter is initialized by an argument in the method call. As you can see in the above loop body, the days parameter decrements down to zero. What happens to the days variable in the main method in Mouse3Driver? Because the two days variables are distinct, the days variable in the main method does not change with the days parameter in the grow method. So when Mouse3Driver prints its version of days, it prints the unchanged value of 365 like this:

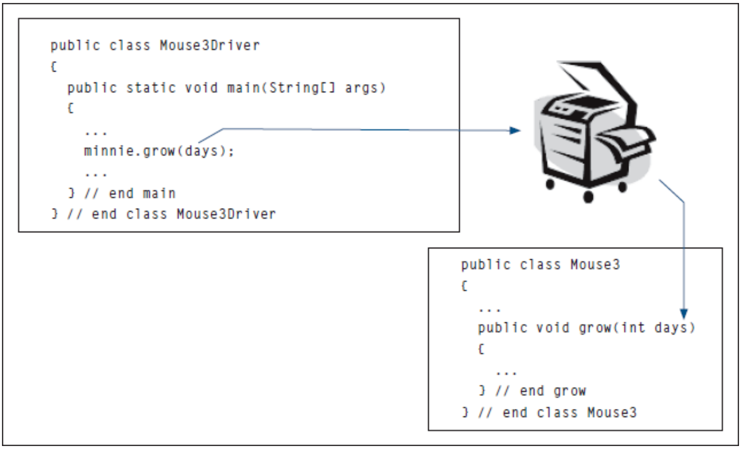


```
# of days aged - 365.
```

Pass-by-Value

We say that Java uses *pass-by-value* for its argument-passing scheme. As illustrated by Figure 6.15, pass-by-value means that the JVM passes a copy of the argument's value (not the argument itself) to the parameter. Changing the copy does not change the original.

In Mouse3Driver and Mouse3, notice that the calling method's argument is called days and the grow method's parameter also is called days. Is the parameter the same variable as the argument? No! They are separate variables separately encapsulated in separate blocks of code. Because these two variables are in separate blocks of code, there is no conflict, and it's OK to give them the same name. Using the same name is natural because these two variables describe the same kind of thing. When names are in different blocks, you don't have to worry about whether they are the same or not. That's the beauty of encapsulation. Big programs would be horrible nightmares if you were prohibited from using the same name in different blocks of code.



```
public class Mouse3Driver
{
  public static void main(String[] args)
  {
    ...
    minnie.grow(days);
    ...
  } // end main
} // end class Mouse3Driver
```

```
public class Mouse3
{
  ...
  public void grow(int days)
  {
    ...
  } // end grow
} // end class Mouse3
```

Figure 6.15 Pass-by-value means a copy of the argument's value goes to the corresponding parameter

Same Name Versus Different Names for Argument-Parameter Pairs

Most of the time, you'll want to use the same name for an argument-parameter pair. But be aware that using different names is legal and fairly common. When it's more natural and reasonable to use different names for an argument-parameter pair, then use different names. The only requirement is that the argument's type must match the parameter's type. For example, in the Mouse3 program, if num is an int variable, then the following method call successfully passes num's value to the days int parameter:

```
minnnie.grow(num);
```

6.12 Specialized Methods—Accessors, Mutators, and Boolean Methods

Let's now discuss some of the common types of specialized methods. You won't be asked to learn any new syntax; you'll just be asked to apply what you've learned so far.

Accessor Methods

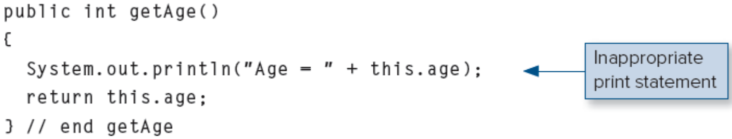
An *accessor* is a method that retrieves part of an object's stored data—

typically private data. Note the following getAge and getWeight methods (taken from Figure 6.14's Mouse2 class). They are accessor methods, as they retrieve the values of the instance variables, age and weight, respectively.

```
public int getAge()
{
  return this.age;
} // end getAge
public double getWeight()
{
  return this.weight;
} // end getWeight
```

As evidenced by the getAge and getWeight methods, accessor methods should be named with a "get" prefix. That's why accessor methods are often called *get methods*.

A method should perform only one task. It should be written so that it accomplishes the one thing its name implies. For example, a getAge method should simply return its object's age instance variable value and do nothing else. We mention this notion because there is sometimes a temptation to provide extra functionality to a method to avoid having to implement that functionality elsewhere. One particularly common *faux pas* (a French term meaning error in etiquette) is to add print statements to a method that doesn't need to print. For example, a novice programmer might implement the getAge method like this:



```
public int getAge()
{
  System.out.println("Age = " + this.age);
  return this.age;
} // end getAge
```

Inappropriate print statement

That getAge method might work fine for the novice programmer's program, which takes into account the getAge method's non-standard print statement. But if another programmer needs to work with the program and call the getAge method later, the new programmer would be surprised to find the non-standard print statement. The new programmer would then either have to (1) accommodate the print statement or (2) remove it from the getAge method and check for any ripple effects. To avoid that scenario, you should include print statements in a method only if the purpose of the method is to print something.

Debug with temporary print statements.

The exception to the above rule is that it's acceptable and helpful to add print statements temporarily to methods when you're trying to debug a program. For example, if you think there's something wrong with your getAge method, you might want to add the above print statement to verify the correctness of the age value just before getAge returns it. If you add such debug print statements, don't forget to remove them later, when your program is working.

Mutator Methods

A *mutator* is a method that changes or “mutates” an object’s state by changing some or all of that object’s stored data —typically private data. For example, here is the mutator method for setting or changing a mouse’s percentGrowthRate instance variable:

```
public void setPercentGrowthRate(double percentGrowthRate)
{
  this.percentGrowthRate = percentGrowthRate;
} // end setPercentGrowthRate
```

As evidenced by the setPercentGrowthRate method, mutator methods should be named with a “set” prefix. That’s why mutator methods are often called *set methods*.

Use mutator to filter input.

An accessor allows you to read a private instance variable. A mutator allows you to update a private instance variable. If you provide a private instance variable with both an accessor and a simple mutator like the setPercentGrowthRate method above, it effectively converts that private instance variable into a public instance variable, and it breaks the encapsulation of that variable. There’s not much danger with having an accessor alone, but having a simple mutator allows an outsider to enter an unreasonable value that may produce erratic program operation. However, if you include constraint checking and perhaps correcting code in your mutators, they can serve as data filters that assign only proper data to your private instance variables. For example, the following setPercentGrowthRate mutator filters out growth rates that are less than −100%:

```
public void setPercentGrowthRate(double percentGrowthRate)
{
  if (percentGrowthRate < -100)
  {
    System.out.println(”Attempt to assign an invalid growth rate.”);
  }
  else
  {
    this.percentGrowthRate = percentGrowthRate;
  }
} // end setPercentGrowthRate
```

Our examples will occasionally include some mutator error checking to illustrate this filtering function, but to reduce clutter, we'll usually employ the minimal form.

Boolean Methods

A *Boolean method* checks to see whether some condition is true or false. If the condition is true, then true is returned. If the condition is false, then false is returned. To accommodate the boolean returned value, Boolean methods must always specify a boolean return type. A Boolean method name should normally start with is. For example, here's an isAdolescent method that determines whether a Mouse object is an adolescent by comparing its age value to 100 days (we're making a simplifying assumption that any mouse younger than 100 days is considered to be an adolescent):

```
public boolean isAdolescent()
{
  if (this.age <= 100)
  {
    return true;
  }
  else
  {
    return false;
  }
} // end isAdolescent
```

Here's how this code might be shortened:

```
public boolean isAdolescent()
{
  return this.age <= 100;
} // end isAdolescent
```

To show how the shortened method works, we'll plug in sample values. But first, let's get settled on the goal: Whenever age is less than or equal to 100, we want the method to return true to indicate adolescence. If age is 50, what is returned? true (This happens because the return statement's this.age < = 100 expression evaluates to true.) If age is 102, what is returned? false (This happens because the return statement's this.age < = 100 expression evaluates to false.) Plug in any number for age and you'll see that the shortened function does indeed work properly. In other words, the shortened isAdolescent method does indeed return true whenever age is less than or equal to 100.

Are you bothered by the lack of parentheses around the return statement's returned expression? With statements that use a condition (if statement, while statement, etc.), the condition must be surrounded by parentheses. With the return statement's returned expression, the parentheses are optional. You'll see it both ways in industry—sometimes parentheses are included and sometimes they're omitted.

Here's how the isAdolescent method

could be used in a calling module:

```
Mouse pinky = new Mouse();
. . .
if (pinky.isAdolescent() == false)
{
  System.out.println(“The mouse’s growth is no longer” +
    ” being simulated - too old.”);
}
```

Do you know how the above if statement can be shortened? Here’s a functionally equivalent if statement with an improved condition:

```
if (!pinky.isAdolescent())
{
  System.out.println("The mouse's growth is no longer" +
    " being simulated - too old.");
}
```

The goal is to print the warning message if pinky is old (not an adolescent). If isAdolescent returns false (indicating an old Pinky), then the if statement’s condition is true (!false evaluates to true) and the program prints the warning message. On the other hand, if isAdolescent returns true (indicating a young Pinky), then the if statement’s condition is false (!true evaluates to false) and the program skips the warning message.

Although the shortened-version if statement might be harder to understand initially, experienced programmers would prefer it. Following that lead, we encourage you to use ! rather than == false for similar situations.

6.13 Problem Solving with Simulation (Optional)

In our previous mouse examples, to keep the focus on OOP concepts rather than mouse growth details, we used a simplistic growth formula. In this section, we show you how to simulate growth in a way that is much closer to the kind of growth that occurs in the real world. Then we show you a simple trick that can be applied to many simulation problems to improve the program's speed and accuracy greatly.

Previously, we modeled growth by assuming that added weight is proportional to weight, like this:

```
addedWeight = fractionGrowthRate × weight
```

where

```
fractionGrowthRate = .01 × percentGrowthRate
```

This kind of growth makes weight increase exponentially and continue to curve upward in time, as indicated by Figure 6.16. This is a good approximation for a young plant or animal, where most of the ingested food energy goes into new growth.

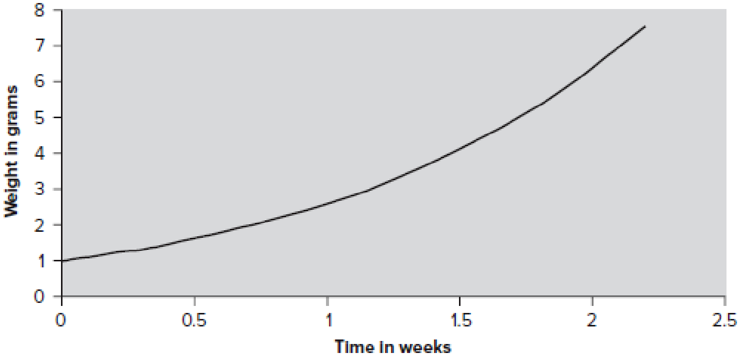


Figure 6.16 Exponential growth

Maturation

But there’s a problem with the exponential growth model. Nothing keeps growing forever! After a while, old tissue starts to die, and some of the ingested nutrients must be used to replace the old tissue instead of just adding to it. This slows the growth. As a larger fraction of ingested nutrients go into replacement, the growth curve straightens out, begins to bend the other way, and approaches a maximum. The easiest way to modify the basic exponential growth formula to make it describe maturation is to multiply by another factor to obtain what’s called the *logistic equation:*

Page 256

$$\mathit{addedWeight} = \mathit{fractionGrowthRate} \times \mathit{weight} \times \left(1.0 - \frac{\mathit{weight}}{\mathit{maxWeight}}\right)$$

A quick inspection of this improved growth formula shows that as weight approaches maxWeight, the quantity in parentheses on the right approaches zero, and therefore the added weight on the left approaches zero. At that point, there’s no more growth. This provides a reasonable description of an organism reaching maturity.

Computer simulations rely on approximate mathematical models, like the model provided by the above logistic equation. Such simulation models are sometimes good, sometimes not so good, and it’s difficult to know how good they are without comparing them to actual live data. But for the current weight gain problem, we have the luxury of being able to compare the simulation model with an exact mathematical model. Here is a closed-form exact mathematical solution that determines the weight of any given time:

$$weight = \frac{1.0}{\frac{1.0}{maxWeight} + e^{-(fractionGrowthRate \times time + g_0)}}$$

This formula contains a growth constant, g_0, which is

$$g_0 = \log_e\left(\frac{minWeight}{1.0 - \frac{minWeight}{maxWeight}}\right)$$

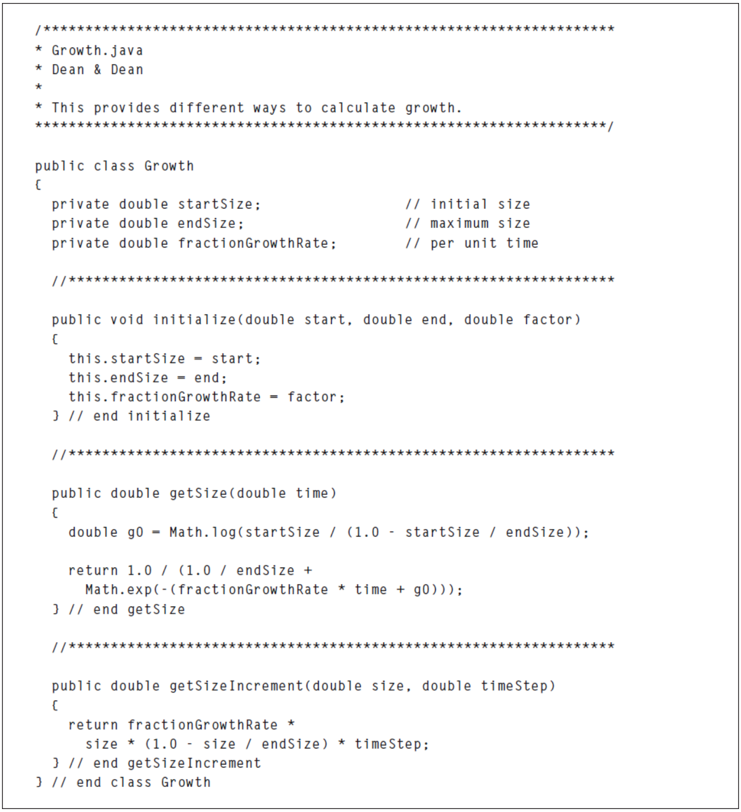
You can find g_0 by plugging

minWeight and *maxWeight* values into the second formula. Then find *weight* by plugging g_0 into the first formula.

Simulation

If you can describe it, you can simulate it.

Usually an exact solution is not available, and the only way to solve a problem is with a simulation. But for this weight gain problem, we have both. Let’s look at a program that displays time, the exact solution, and the simulated solution together. See the program’s Growth class in Figure 6.17.



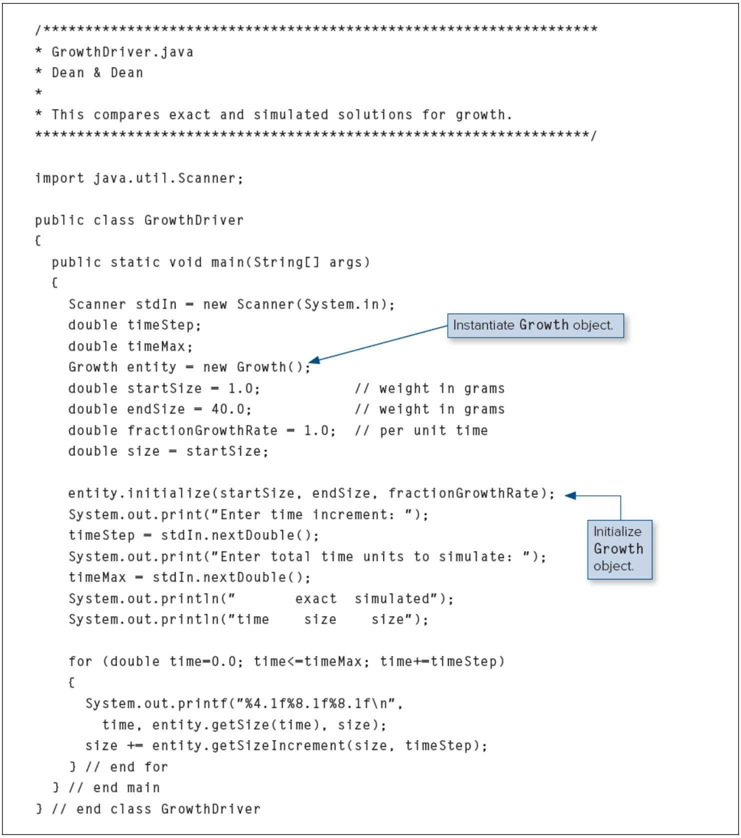
```
/*********************************************************************
* Growth.java
* Dean & Dean
*
* This provides different ways to calculate growth.
*********************************************************************/

public class Growth
{
  private double startSize;              // initial size
  private double endSize;                // maximum size
  private double fractionGrowthRate;     // per unit time

  //*******************************************************************

  public void initialize(double start, double end, double factor)
  {
    this.startSize = start;
    this.endSize = end;
    this.fractionGrowthRate = factor;
  } // end initialize

  //*******************************************************************

  public double getSize(double time)
  {
    double g0 = Math.log(startSize / (1.0 - startSize / endSize));

    return 1.0 / (1.0 / endSize +
      Math.exp(-(fractionGrowthRate * time + g0)));
  } // end getSize

  //*******************************************************************

  public double getSizeIncrement(double size, double timeStep)
  {
    return fractionGrowthRate *
      size * (1.0 - size / endSize) * timeStep;
  } // end getSizeIncrement
} // end class Growth
```

Figure 6.17 Growth class, which implements different ways to evaluate growth

The Growth class has three instance variables, startSize, endSize, and fractionGrowthRate, and three methods. The initialize method initializes the three instance variables. The getSize method uses the closed-form mathematical solution formula provided earlier. It returns the size (e.g., current mouse weight) at the given time. Notice that this method's name starts with "get," so it looks like the name of an accessor method, and it returns a double value just as our previous getWeight method does. But this class does not have any instance variable named "size." So here's an example of a method that is not really an accessor like the accessors described in Section 6.12, even though its name makes it look like an accessor. The point is this: any method can return a value, not just an accessor method, and any method can have any name that seems appropriate—getSize is simply the most appropriate name we could think of for this method that computes and returns a size.

The getSizeIncrement method implements one simulation step. It returns the change in size between the current time and the next time. Notice that the getSize and getSizeIncrement methods do different things. The first one gives the answer directly. The second one gives an incremental value that must be added to a previous answer to get the next answer.

If you are writing your own class and you want to model the growth of one of your class's entities, you could copy and paste the Growth class's variables and methods into your class. Alternatively, you could delegate the work to a Growth class object just as you delegate work to Scanner class objects. To do this, use new to instantiate a Growth object, initialize it with the growth-related data in your object. Then ask the Growth object to solve the growth problem for you by calling its getSize or getSizeIncrement method. In your program, you could use code like that in the main method of the GrowthDriver class in Figure 6.18.



```
/*******************************************************************
* GrowthDriver.java
* Dean & Dean
*
* This compares exact and simulated solutions for growth.
*******************************************************************/

import java.util.Scanner;

public class GrowthDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double timeStep;
    double timeMax;
    Growth entity = new Growth();
    double startSize = 1.0;           // weight in grams
    double endSize = 40.0;            // weight in grams
    double fractionGrowthRate = 1.0;  // per unit time
    double size = startSize;

    entity.initialize(startSize, endSize, fractionGrowthRate);
    System.out.print("Enter time increment: ");
    timeStep = stdIn.nextDouble();
    System.out.print("Enter total time units to simulate: ");
    timeMax = stdIn.nextDouble();
    System.out.println("       exact  simulated");
    System.out.println("time    size    size");

    for (double time=0.0; time<=timeMax; time+=timeStep)
    {
      System.out.printf("%4.1f%8.1f%8.1f\n",
        time, entity.getSize(time), size);
      size += entity.getSizeIncrement(size, timeStep);
    } // end for
  } // end main
} // end class GrowthDriver
```

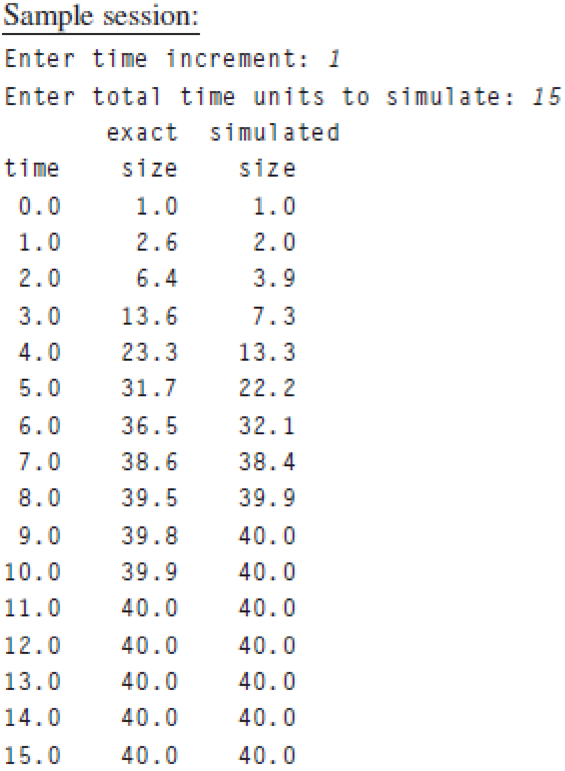
Instantiate `Growth` object.

Initialize `Growth` object.

Figure 6.18 GrowthDriver class that demonstrates the Growth class in Figure 6.17

Page 258

This driver class may seem imposing, but it's not difficult. We start by declaring and initializing local variables, and this includes instantiating and initializing a Growth object. Then we ask the user to provide a time increment and the total number of time increments. Finally, we use a for loop to print time, the exact solution, and the simulated solution for each time step. If you run the program composed of the code in Figures 6.17 and 6.18, you'll get this result:



Sample session:

```
Enter time increment: 1
Enter total time units to simulate: 15
       exact  simulated
time    size    size
 0.0     1.0     1.0
 1.0     2.6     2.0
 2.0     6.4     3.9
 3.0    13.6     7.3
 4.0    23.3    13.3
 5.0    31.7    22.2
 6.0    36.5    32.1
 7.0    38.6    38.4
 8.0    39.5    39.9
 9.0    39.8    40.0
10.0    39.9    40.0
11.0    40.0    40.0
12.0    40.0    40.0
13.0    40.0    40.0
14.0    40.0    40.0
15.0    40.0    40.0
```

Figure 6.19 shows what this data looks like in a two-dimensional plot. Alas, the simulated solution doesn't agree very well with the exact solution. It doesn't rise quickly enough, and then it overshoots. The reason for this error is actually quite simple. Each size increment is based on the size at the beginning of the increment. But as time passes, the actual size changes, so for all but the first instant in the increment, the calculation is using old data.

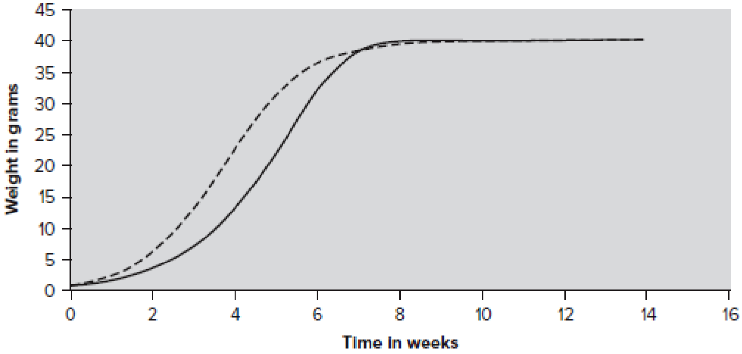


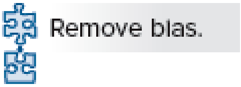
Figure 6.19 Simulated solution with time increment = 1 (solid), compared to the exact solution (dashed)

The most straightforward way to fix this accuracy problem is to use a smaller time step. With this simulation algorithm, the error is proportional to the size of the time step. If you cut the time step in half, this cuts the error in half, if you divide the time step by 10, this divides the error by 10, and so on. In the above output, at four weeks, the exact solution says the size is 23.3 grams, but the simulation says it’s only 13.3 grams. That’s an error of $23.3 - 13.3 = 10$ grams. If we want to reduce this error to less than 1 gram, we need to reduce the time step by a factor of about 10.

If you don’t know the exact solution, how do you know your error? Here’s a rule of thumb: If you want less than 1% error, make sure the size increment in each time step is always less than about 1% of the average size in that time interval.

⚠ This simple algorithm works fine for simple problems. But if you have a tough problem, some things may be sensitive to very small errors, and you may have to take a very large number of very small steps. This might take more time than you can stand. There's also a more insidious problem. Even a double number has limited precision, and when you process many numbers, round-off errors can accumulate. In other words, as you make step sizes smaller, errors initially decrease, but eventually they begin to increase again.

Improved Accuracy and Efficiency Using a Step-with-Midpoint Algorithm[3]

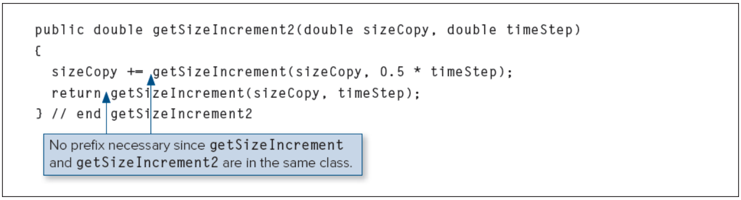


Remove bias.

There's a better way to improve accuracy. It's based on a simple principle: Instead of using the condition(s) (e.g., weight) at the beginning of the interval to estimate the change(s) during the interval, use the condition(s) in the middle of the interval to estimate the change(s) during the interval. But how can you know the conditions in the middle of the interval until you get there? Send out a “scouting party”! In other words, make a tentative half-step forward, and evaluate the conditions there. Then go back to the beginning and use the condition(s) at the midpoint to determine what the change(s) will be in a full step forward.

🚲 At first, this might sound like a hard way to do an easy thing. Why not just cut the step size in half and take two small steps forward? The qualitative answer is: That still leaves a regular bias toward old data. The quantitative answer is: If you use a step-with-midpoint algorithm for your simulation, the size of the error is proportional to the square of the size of the time step. That means that if you reduce the full-step size by a factor of 100, the error goes down by a factor of 10,000. In other words, you can get an extra factor-of-100 accuracy by increasing the computer's work by only a factor of 2.

But what about the work you do? How much harder is it to implement a step-with-midpoint algorithm? Not much. All you have to do is add one simple method. Specifically, to the Growth class in Figure 6.17, just add the getSizeIncrement2 method shown in Figure 6.20.



```
public double getSizeIncrement2(double sizeCopy, double timeStep)
{
  sizeCopy += getSizeIncrement(sizeCopy, 0.5 * timeStep);
  return getSizeIncrement(sizeCopy, timeStep);
} // end getSizeIncrement2
```

No prefix necessary since getSizeIncrement and getSizeIncrement2 are in the same class.

Figure 6.20 A method that implements the step-with-midpoint algorithm Add this method to the code in Figure 6.17 to improve simulation accuracy and efficiency.

Page 261

How does this little method work? It simply calls the original getSizeIncrement method two times. Notice that the sizeCopy parameter in

Figure 6.20 is just a copy of the size variable in the driver class. The first call to getSizeIncrement uses the size at the beginning of the time increment, and it goes only half a time step forward. Then, it uses the returned value to increment sizeCopy to the size at the midpoint. The second call to getSizeIncrement uses this computed midpoint size and a full time step to determine the change from the beginning to the end of the full time interval.

Within the getSizeIncrement2 method definition, note the calls to getSizeIncrement. There's no reference variable dot prefix at the left of getSizeIncrement. Here's why: If you call a method that's in the same class as the current class, then you can call the method directly, with no reference variable dot prefix.

The work required to modify the driver is negligible. All you have to do is change the name of the method called to the name of the new method. In our case, all you have to do is change the last statement in the driver in Figure 6.18 to this:

```
size += entity.getSizeIncrement2(size, timeStep);
```

This appended “2” is the only difference!

Figure 6.21 shows what the improved algorithm produces with a full step size equal to the step size used for Figure 6.19. This takes twice as much computer time as what’s in Figure 6.19, but it’s clearly much more than twice as good. For example, at four weeks, the error is now only 1.5 grams instead of the previous 10 grams.

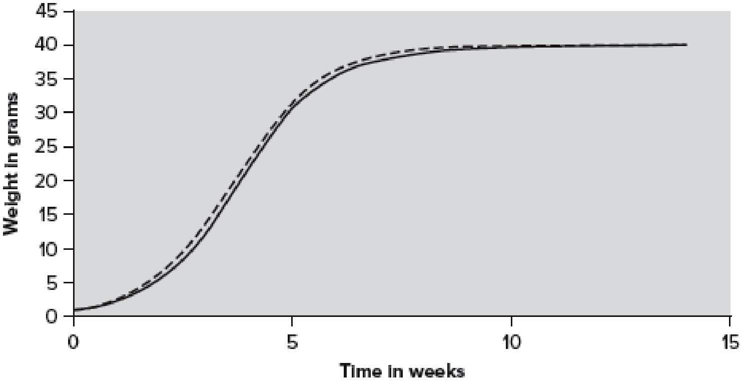


Figure 6.21 Step-with-midpoint

simulated solution with time increment = 1 (solid), compared to the exact solution (dashed)

Summary

- An object is a group of related data that identifies the current condition or *state* of the object, plus the methods that describe the *behavior* of that object.
- Objects are *instances* of the classes that define them. A class definition specifies the instance variables which an object of that class contains, and it defines the methods which an object of that class may call. Each object contains its own copy of the instance variables its class defines, and a given instance variable generally has different values in different objects.

- Use the private access modifier to specify that a particular variable is encapsulated or hidden. Use the public access modifier to make methods accessible to the outside world.

- To make a class as general as possible, drive it from a main method in a separate “driver” class. In the driver’s main method, declare a reference variable of the driven class’s type. Then, use Java’s keyword new to instantiate an object of the driven class, and initialize the reference variable with the object reference returned by new.

- Use Java’s keyword this to refer to the calling object from within one of that object’s methods. Use this to distinguish an instance variable from a same-named parameter or local variable.

- When you trace an object-oriented program, you need to keep track of which class you're in, which method you're in, which object called that method, parameter and local variable names, and the names of all instance variables in each object.

- A UML class diagram has separate boxes for the class name, a description of the class's variables, and headings for the class's methods. Use a "+" prefix for public and a "-" prefix for private. Specify variable and method return types and non-default initial values.

- Instance variable default values are zero for numbers, false for boolean values, and null for references. Instance variable values persist for the life of their object. Local variable default values are undefined garbage. Local variables and parameters persist for as long as their method is being executed, and after that, their values are undefined.

- Unless a method’s return type is void, every path through the method must end with a statement that returns a value of the method’s type.

- A method’s parameter must have the same type as the method call’s argument. What the method gets is a copy of what is in the calling program, so changing a parameter in a method does not change the calling program’s value.

- Use setX and getX methods to modify and retrieve private instance variable values. Include filtering in setX methods to protect your program from bad input. Use boolean isX methods to return true or false, depending on the value of a particular condition.

- Optionally, improve simulation speed and accuracy by computing the next increment with values determined halfway between that increment's starting and ending points.

Review Questions

---

§6.2 Object-Oriented Programming Overview

1. A class is an instance of an object. (T / F)

2. How many objects may there be within a single class?

§6.3 First OOP Class

3. A class's instance variables must be declared outside of all

_____, and all instance variable declarations should be located at the _______________.

4. Methods accessible from outside a class are public, but instance variables (even those that an outsider may need to change or read) are usually private. Why?

§6.4 Driver Class

5. Where does main go—in the driver class or in one of the driven classes?

6. When a program has both driver and driven classes, where should most of the program code reside?

7. How do you retrieve a private instance variable's value from within a main method?

8. A reference variable holds the ____________ of an object.

§6.5 Calling Object, this Reference

9. An instance method might contain a statement like this.weight = 1.0; but if that method's class currently has five instantiated objects, there are five different variables called weight. How can we determine which one is getting the new value?

§6.6 Instance Variables

10. What are the default values for int, double, and boolean for an object's instance variables?

11. In the Mouse program of Figures 6.4 and 6.5, what is the persistence of gus's age variable?

§6.8 UML Class Diagrams

12. After a program is written, a UML class diagram provides a brief outline of each class in the program. It helps other people see what methods are available and what arguments they need. Give some reasons why it might be helpful to have an already created class diagram in front of you while you are implementing the class and writing its methods.

§6.9 Local Variables

13. Assume the main method in Mouse2Driver had started more simply with only Mouse mickey; What would be the value of mickey immediately after this statement?

§6.10 The return Statement

14. Usually, the use of multiple return statements leads to code that is more understandable. (T / F)

§6.11 Argument Passing

15. How is a method parameter like a local variable, and how do they differ?

16. What is the relationship and difference between a method argument and a method parameter?

§6.12 Specialized Methods—Accessors, Mutators, and Boolean Methods

17. What is the standard prefix for an accessor method?

18. What is the standard prefix for a mutator method?

19. What is the standard prefix for a Boolean method?

§6.13 Problem Solving with Simulation (Optional)

20. Identify two general ways to reduce the size of the error in a simulation. For a given accuracy, which way is more efficient?

Exercises

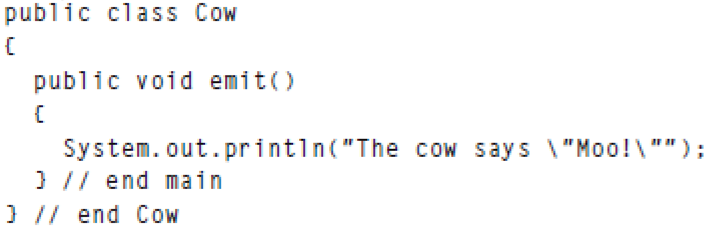
---

1. [after §6.2] Suppose you are asked to model animals using an OOP program. For each of the following animal-related entities, select one of the following object-oriented programming constructs: class, object, instance variable, or method.

- **a)** whether the animal has a backbone
- **b)** animal weight
- **c)** an individual animal
- **d)** sequence of activities that occur when some stranger approaches

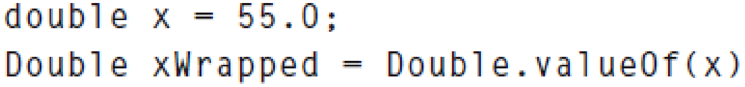
2. [after §6.2] What are the two kinds of static members?

3. [after §6.4] Given the following Cow class. Provide a complete CowDriver class that calls the emit method.



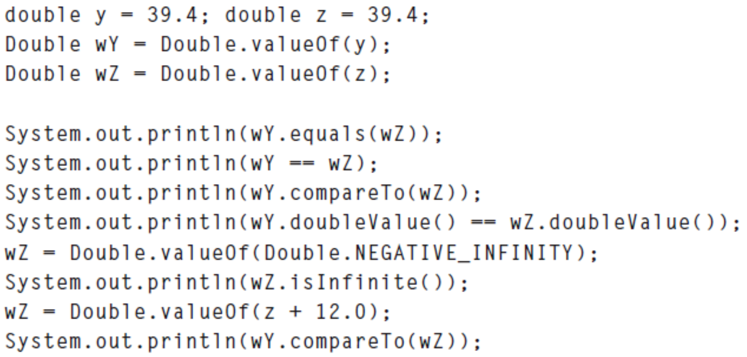
```
public class Cow
{
  public void emit()
  {
    System.out.println("The cow says \"Moo!\"");
  } // end main
} // end Cow
```

4. [after §6.4] Wrapper objects: The wrapper classes discussed in Chapter 5 also provide you with the ability to instantiate objects that are wrapped versions of primitive variables. For example, to create a wrapped version of the double number x, you can do this:



```
double x = 55.0;
Double xWrapped = Double.valueOf(x)
```

This creates an object of type Double, which is a wrapped version of the primitive variable, x. Then it assigns a reference to that object to the reference variable, xWrapped. The Double class has a number of prebuilt methods that work with Double objects. You can read about these methods in the Java API documentation on the Double class. The following code illustrates some of these methods:



```
double y = 39.4; double z = 39.4;
Double wY = Double.valueOf(y);
Double wZ = Double.valueOf(z);

System.out.println(wY.equals(wZ));
System.out.println(wY == wZ);
System.out.println(wY.compareTo(wZ));
System.out.println(wY.doubleValue() == wZ.doubleValue());
wZ = Double.valueOf(Double.NEGATIVE_INFINITY);
System.out.println(wZ.isInfinite());
wZ = Double.valueOf(z + 12.0);
System.out.println(wY.compareTo(wZ));
```

What does each of the above print statements output, and why?

5. [after §6.6] Suppose you own some land and would like to build three houses on it.

a) Implement a House class with instance variables floorArea (a double) and salePrice (another double) and these methods:

i. A build method with a double floorArea parameter. This method should assign its parameter to the corresponding instance variable and print the value of its parameter.

ii. A sell method that initializes the salePrice instance variable with 150 times the value of the floorArea instance variable and then prints out the value of this salePrice.

b) Implement a Housing class with a main method that does the following:

i. Declare three variables—house (a reference to a House object), floorArea (a double), and totalFloorArea (another double, initialized to 0).

ii. Execute a for loop with three iterations. In each iteration, do this:

a. Assign a random value between 1000 and 2500 to floorArea.

b. Instantiate a House object and assign the result to house.

c. Call House’s build method with floorArea as argument.

d. Add floorArea to totalFloorArea.

e. Calls House’s sell method.

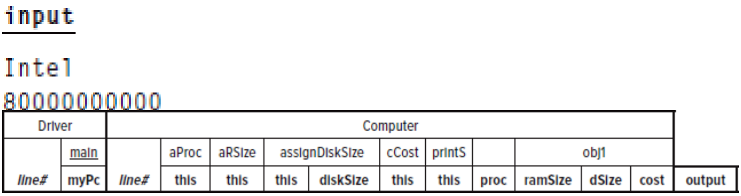
iii. Print the accumulated value in totalFloorArea.

6. [after §6.7] Given this Computer design program:

```
/********************************************************
* ComputerDriver.java
* Dean & Dean
*
* This exercises the Computer class.
********************************************************/

public class ComputerDriver
{
  public static void main(String[] args)
  {
    Computer myPc = new Computer();
    myPc.assignProcessor();
    myPc.assignRamSize();
    myPc.assignDiskSize();
    myPc.calculateCost();
    myPc.printSpecification();
  } // end main
} // end class ComputerDriver
```

```
/***********************************************************
* Computer.java
* Dean & Dean
*
* This class collects specifications for a Computer.
***********************************************************/

import java.util.Scanner;

public class Computer
{
  private String processor;
  private long ramSize = (long) 1000000000.0;
  private long diskSize;
  private double cost;

  //********************************************************

  public void assignProcessor()
  {
    Scanner stdIn = new Scanner(System.in);
    this.processor = stdIn.nextLine();
  } // end assignProcessor

  //********************************************************

  public void assignRamSize()
  {
    this.ramSize = (long) 3000000000.0;
  } // end assignRamSize

  //*******************************************************

  public void assignDiskSize()
  {
    Scanner stdIn = new Scanner(System.in);
    long diskSize;
    diskSize = stdIn.nextLong();
  } // end assignDiskSize

  //*******************************************************

  public void calculateCost()
  {
    this.cost = this.ramSize / 20000000.0 +
      this.diskSize / 200000000.0;
    if (this.processor.equals(”Intel”))
    {
      this.cost += 200;
    }
    else
    {
      this.cost += 150;
    }
  } // end calculateCost

  //*******************************************************

  public void printSpecification()
  {
    System.out.println(”Processor = ” + this.processor);
    System.out.println(”RAM = ” + this.ramSize);
    System.out.println(”Hard disk size = ” + this.diskSize);
    System.out.println(”Cost = $” + this.cost);
  } // end printSpecification
} // end class Computer
```

Use the following trace setup to trace the Computer program. Note that we have used abbreviations to keep the trace setup’s width as small as possible. Don’t forget to specify default and initial values, even if they don’t affect the final result.



```
input

Intel
80000000000
```

Driver		Computer											
	main		aProc	aRSize	assignDiskSize		cCost	printS		obj1			
line#	myPc	*line#*	this	this	this	diskSize	this	this	proc	ramSize	dSize	cost	output

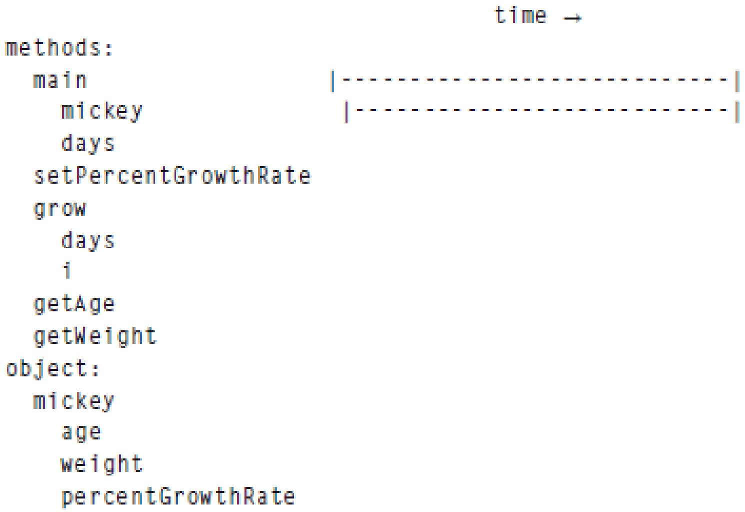
7. [after §6.8] What does UML mean? That is, what words do its initials represent?

8. [after §6.8] Construct UML class diagrams for the Computer and ComputerDriver classes in Exercise 6.

9. [after §6.9] When a method's execution terminates, its local variable values become garbage. (T / F)

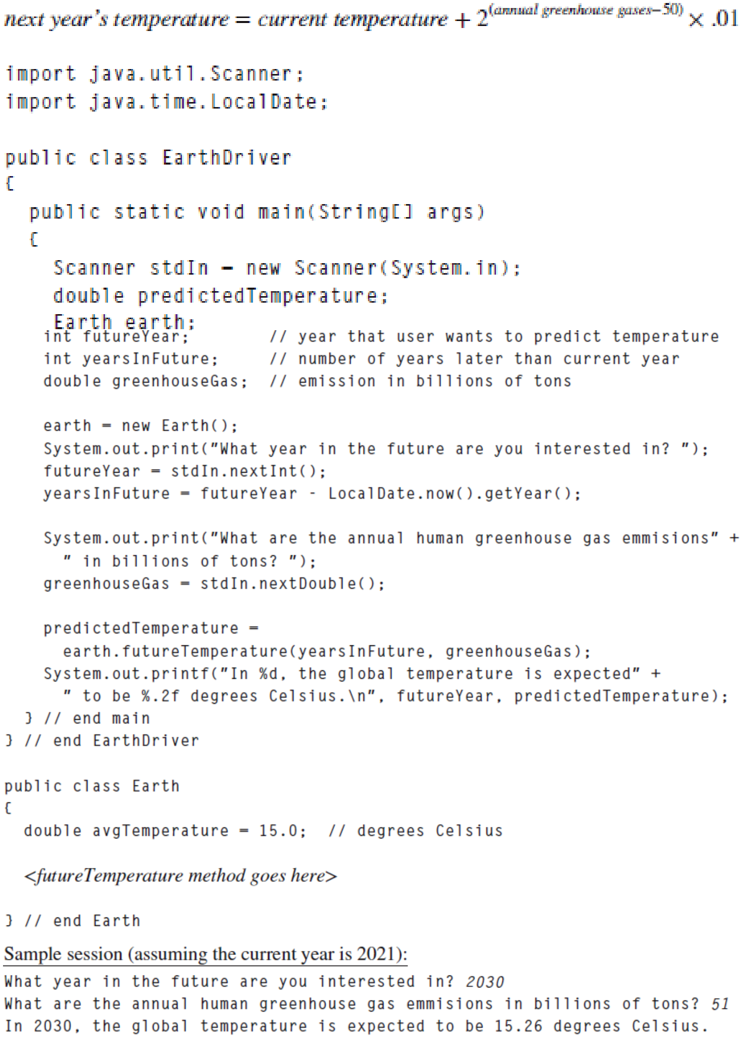
Page 267

10. [after §6.11] The diagram below shows Mouse2 program methods, with their parameters and local variables indented, and the one instantiated object, with its instance variables indented. Your task is to construct a time line for each method, local variable or parameter, object, and instance variable. Each time line should show that item's persistence (when it starts and ends) relative to the other items. To help you get started, we have provided the time lines for the main method and one of its local variables. Provide all the other time lines, and show how they align with each other and those already provided. (Assume that the object and its instance variables come into existence simultaneously.)



```
                                   time →
methods:
  main                      |-----------------------------|
    mickey                   |----------------------------|
    days
  setPercentGrowthRate
  grow
    days
    i
  getAge
  getWeight
object:
  mickey
    age
    weight
    percentGrowthRate
```

11. [after §6.11] Given the following Earth class and skeleton for an EarthDriver class, provide a futureTemperature method that receives two parameters, years and annualGreenhouseGases. The years parameter holds the number of years into the future that the user is interested in. The annualGreenhouseGases parameter holds the annual emission of human-produced greenhouse gases in billions of tons. The method returns the predicted temperature for the given number of years into the future. In your method, use this (completely fabricated and unscientific) formula to calculate the earth’s temperature one year into the future:



$$\textit{next year's temperature} = \textit{current temperature} + 2^{(\textit{annual greenhouse gases} - 50)} \times .01$$

```
import java.util.Scanner;
import java.time.LocalDate;

public class EarthDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double predictedTemperature;
    Earth earth;
    int futureYear;         // year that user wants to predict temperature
    int yearsInFuture;      // number of years later than current year
    double greenhouseGas;   // emission in billions of tons

    earth = new Earth();
    System.out.print("What year in the future are you interested in? ");
    futureYear = stdIn.nextInt();
    yearsInFuture = futureYear - LocalDate.now().getYear();

    System.out.print("What are the annual human greenhouse gas emmisions" +
      " in billions of tons? ");
    greenhouseGas = stdIn.nextDouble();

    predictedTemperature =
      earth.futureTemperature(yearsInFuture, greenhouseGas);
    System.out.printf("In %d, the global temperature is expected" +
      " to be %.2f degrees Celsius.\n", futureYear, predictedTemperature);
  } // end main
} // end EarthDriver

public class Earth
{
  double avgTemperature = 15.0;  // degrees Celsius

  <futureTemperature method goes here>

} // end Earth
```

Sample session (assuming the current year is 2021):

```
What year in the future are you interested in? 2030
What are the annual human greenhouse gas emmisions in billions of tons? 51
In 2030, the global temperature is expected to be 15.26 degrees Celsius.
```

12. [after §6.12] Complete the following VoterDriver class skeleton by replacing all six occurrences of <*insert code here*> with appropriate code such that the program operates properly. For details, read the comments above and next to the <*insert code here*> items. Note the Voter class, which is below the VoterDriver class. The two classes are in separate files.

```
import java.util.Scanner;

public class VoterDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Voter voter;  // Voter object
    String name;  // voter name

    // instantiate a Voter object and assign it to voter.
    <insert code here>

    System.out.print("Enter voter name: ");
    name = stdIn.nextLine();
    // Assign name to the Voter object.
    <insert code here>

    System.out.print("Enter voter id: ");
    // In a single statement, read an int for the id value
    // and assign it to the Voter object.
    <insert code here>

    // If invalid id, execute the loop.
    // (Use the isValid method in the while loop condition.)
    while (<insert code here>)
    {
      System.out.print("Invalid voter id - reenter: ");
      // In a single statement, read an int for the id value
      // and assign it to the Voter object.
      <insert code here>
    }

    System.out.println("\n" + name +
      ", your new e-mail account is: \n" +
      <insert code here>             // get email account.
  } // end main
} // end class VoterDriver

public class Voter
{
  private String name;
  private int id;

  //*********************************************************

  public void setName(String n)
  {
    this.name = n;
  }

  public void setId(int id)
  {
    this.id = id;
  }

  //*********************************************************

  public String getEmailAccount()
  {
    // Include "" in concatenation to convert to strings.
    return "" + this.name.charAt(0) + this.id +
      "@voters.org";
  }
  //*********************************************************

  public boolean isValid()
  {
    return this.id >= 100000 && this.id <= 999999;
  }
} // end class Voter
```

13. [after §6.13] Change the Growth and GrowthDriver classes in Section 6.13 as described in the Improved Accuracy and Efficiency Using a Step-with-Midpoint Algorithm subsection. Run your modified program with these inputs:

```
time increment: 1
total time units to simulate: 14
```

For your answer, display the 14 rows of output numbers with each row showing time, exact size, and simulated size.

Review Question Solutions

1. False. An object is an instance of a class.

2. Any number, including zero.

3. A class's instance variables must be declared outside of all methods, and all instance variable declarations should be located at the top of the class definition.

4. Instance variables are usually private to further the goal of encapsulation. That means an object's data is harder to access (and, consequently, harder to mess up). The only way for the data to be accessed from outside of the class is if the data's associated public methods are called.

5. The main method goes in the driver class.

6. Most of a program's code should be in driven classes.

7. To access a private instance variable from within a main method, you have to use an instantiated object's reference variable and then call an accessor method. In other words, use this syntax:

reference-variable.accessor-method-call

8. A reference variable holds the memory location of an object.

9. Go back to where the method was called, and look at the reference variable that precedes the method name at that point. That reference variable is the one that the method uses whenever this is used.

10. For an object's instance variables, the default values are: int = 0, double = 0.0, boolean = false.

11. gus's age is an instance variable. Instance variables persist for the duration of a particular object. Because the gus object is declared in main, gus and its instance variables (including age) persist for the duration of the main method.

12. Some reasons to construct a UML class diagram before writing code are the following:

a) It provides a complete "to do" list. When you are into the details of writing one method, and wondering whether that method should perform a particular function, the diagram reminds you of what other methods might be able to perform that function.

b) It provides a complete "parts list," like the parts list of a typical user-assembled "kit." This predefined list helps you avoid accidentally generating different and conflicting names for variables and parameters as you write your code.

c) It’s a working document that can change as work progresses. Changing the UML class diagram helps identify needed alterations to previous work.

13. Immediately after the statement Mouse mickey; the value of mickey would be garbage.

14. False. Normally, for a method that returns a value, you should have a single return statement at the end of the method. However, it’s also legal to have return statements in the middle of a method. That might be appropriate in a very short method, where an internal return is immediately obvious. If the method is relatively long, however, a reader might not notice an internal return. With a large method, it's better practice to arrange things so that there is only one return, located at the end of the method.

15. Parameters and local variables both have method scope and persistence. The code inside the method treats parameters just like it treats local variables. The method initializes the local variables, while the method call initializes the parameters.

16. Arguments and parameters are two different words describing data that passes into a called method. An arguments is the method call's name for the data, and a parameter is the method's name for the same data. A parameter is just a copy of the method call's argument, however, so if the called method changes the value of one if its parameters, this does not alter the value of the method call's argument.

17. The standard prefix for an accessor method is get.

18. The standard prefix for a mutator method is set.

19. The standard prefix for a Boolean method is is.

20. To reduce the error in a simulation, you can reduce step size or switch to a step-with-midpoint algorithm. For a given accuracy, the step-with-midpoint algorithm is more efficient.

---

[1] Gus and Jaq are mice in the Disney classic *Cinderella.*

[2] We're keeping things simple by telling you to put both classes in the same directory. Actually, the files may be in different directories, but then you'd need to use a package to group together your classes. Appendix 4

describes how to group classes into a package.

[3] The formal name for this algorithm is "Second-order Runge-Kutta."

Object-Oriented Programming—Additional Details

Objectives

- Improve your understanding of the relationship between a reference variable and an object.
- Learn what happens when you

assign a reference.

- Learn how Java recycles memory space.
- Learn how to compare the equality of two different objects.
- Be able to swap the data in two different objects.
- See how a reference parameter can enhance data transfer to and from a called method.
- Learn how to execute a sequence of several method calls in the same statement.
- Learn how to create alternative variations for a method.
- Learn how to combine object creation and initialization in a constructor.
- Learn how to avoid code redundancy by calling a constructor from another

constructor.

- Learn how and when to use static variables.
- Learn how to write static methods and when to use them.
- Learn how and when to use static constants.

Outline

7.1 Introduction 273

7.2 Object Creation—A Detailed Analysis 273

7.3 Assigning a Reference 275

7.4 Testing Objects for Equality 279

7.5 Passing References as Arguments 284

7.6 Method-Call Chaining 286

7.7 Overloaded Methods 289

7.8 Constructors 293

7.9 Overloaded Constructors 299

7.10 Static Variables 303

7.11 Static Methods 306

7.12 Named Constants 312

7.13 Problem Solving with Multiple Driven Classes 314

7.1 Introduction

In Chapter 6, you learned to write simple object-oriented programming (OOP) programs using simple OOP building blocks. In this chapter, you learn to write more advanced OOP programs using more advanced OOP concepts. In particular, you learn the details of what happens behind the scenes when a program instantiates an object and stores its address in a reference variable. That will help you to appreciate and understand what happens when a program assigns one reference variable to another.

One of the OOP concepts you learn about in this chapter is testing objects for equality. It's common to compare primitives for equality. For example, `if (team1Score == team2Score)` asks whether two variables have the same value. It's also common to compare references for equality. That means we're asking whether two references refer to the same object. Comparing references for equality requires a bit more effort, and in this chapter, you learn what that effort entails. Another concept you learn about is what happens behind the scenes when a program passes a reference as an argument. That's important to know because you'll often need to pass references as arguments.

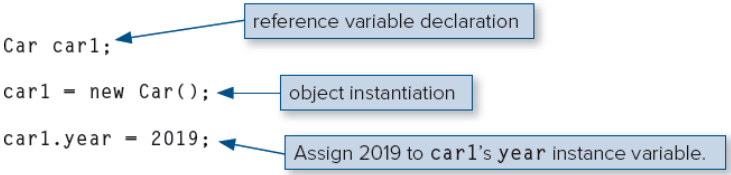
In addition to presenting more advanced OOP concepts, this chapter also presents more advanced applications of what you already know in regard to OOP. For example, you learn to call several methods in succession, all within one statement. That's called *method-call chaining,* and it can lead to more compact and more elegant code. You also learn about *method overloading.* That's when you have different versions of a method, and each version operates on different types of data. That should sound familiar because you saw it with the `Math` class. Remember the two versions of the `Math.abs` method? One version returns the absolute value of a `double`, and one version returns the absolute value of an `int`.

In the previous chapter, you learned how to instantiate an object in one statement (e.g., `Mouse gus = new Mouse();`) and assign a value to the object in a separate statement (e.g.,`gus.setPercentGrowthRate(10);)` In this chapter, you learn how to combine those two tasks into one statement. To do that, you'll use a special kind of method called a *constructor*. Like methods, constructors can be overloaded by using different types of data for the different constructor versions. But unlike methods, constructors are designed specifically for object creation and initialization.

7.2 Object Creation—A Detailed Analysis

Let's start the chapter with a behind-the-scenes detailed look at what happens when a program instantiates an object and stores its address in a reference variable. Having a clear understanding will help when it comes time to understand other OOP operations, and it will help with some debugging efforts.

Consider the following code fragment:



```
Car car1;
car1 = new Car();
car1.year = 2019;
```

reference variable declaration

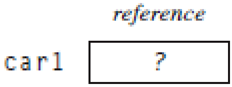
object instantiation

Assign 2019 to `car1`'s `year` instance variable.

Let's now examine this code in detail one statement at a time.

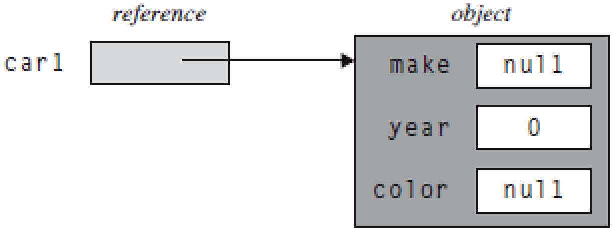
Statement 1:

The first statement is a variable declaration for the `car1` reference variable. It allocates space in memory for the `car1` reference variable—just the reference variable itself, not an object. Eventually, the `car1` reference variable will hold the address of an object, but since there's no object created for it yet, it doesn't yet hold a legitimate address. What's the default value for a reference variable? It depends. If the reference variable is defined locally within a method (that is, it's a local variable), then it gets garbage initially. If it's defined at the top of a class, above all the method definitions (that is, it's an instance variable), then it gets initialized to `null`. Because Statement 1 doesn't have an access modifier like `public` or `private`, it's reasonable to assume it's a local variable. So `car1` will contain garbage by default, and that's what this picture indicates:



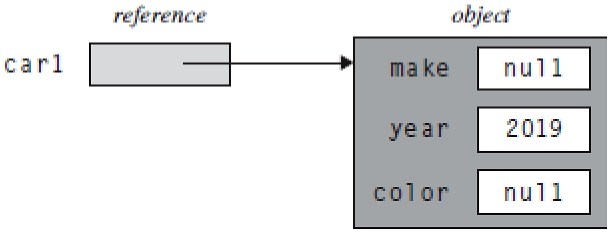
Statement 2:

The second statement's `new` operator allocates space in memory for a new `Car` object, and it initializes members of that new object as specified by the `Car` class definition. The assignment operator assigns the address (memory location) of the allocated space to the `car1` reference variable. Don't forget this operation. Forgetting to instantiate is a common beginner's error.



Statement 3:

The third statement uses the `car1` variable's value (the address of a `Car` object) to find a particular `Car` object in memory. Once that `Car` object is found, 2019 is assigned into it. More specifically, 2019 is assigned to the `year` instance variable portion of that `Car` object. Normally, we'd use a method to assign 2019 to `car1`'s `year` instance variable. In the interest of simplification for clarity's sake, we avoided the method call by assuming that `year` is a `public` instance variable.

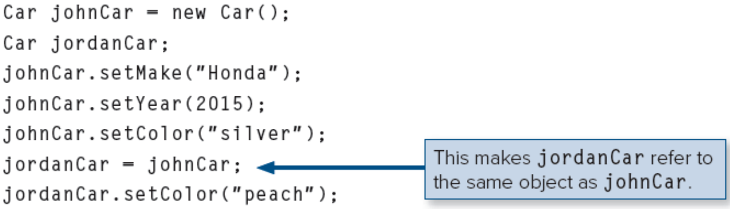


7.3 Assigning a Reference

The result of assigning one reference variable to another is that both reference variables then refer to the same object. Why do they refer to the same object? Because reference variables store addresses, you're actually assigning the right-side reference variable's address into the left-side reference variable. So after the assignment, the two reference variables hold the same address, and that means they refer to the same object. With both reference variables referring to the same object, if the object is updated using one of the reference - variables, then the other reference variable will benefit (or suffer) from that change when it attempts to access the object. Sometimes that's just what you want, but if it's not, it can be disconcerting.

An Example

Suppose you want to create two `Car` objects that are the same except for their color. Your plan is to instantiate the first car, use it as a template when creating the second car, and then update the second car's `color` instance variable. Will this code accomplish that?



```
Car johnCar = new Car();
Car jordanCar;
johnCar.setMake("Honda");
johnCar.setYear(2015);
johnCar.setColor("silver");
jordanCar = johnCar;
jordanCar.setColor("peach");
```

This makes `jordanCar` refer to the same object as `johnCar`.

The problem with the above code is that the `jordanCar = johnCar;` statement causes the two references to point to the same single `Car` object. Figure 7.1a illustrates what we're talking about.

Later, we'll see that this *aliasing* (using different names for the same object) can be quite useful, but in this case, it's not what we wanted. In the last statement in the code fragment above, when we use the `setColor` method to change Jordan's car to "peach," we're not specifying the color for a new car. What we're doing is repainting the original car. Figure 7.1a depicts the result. Uh oh . . . John may not be pleased to find his car repainted to peach!

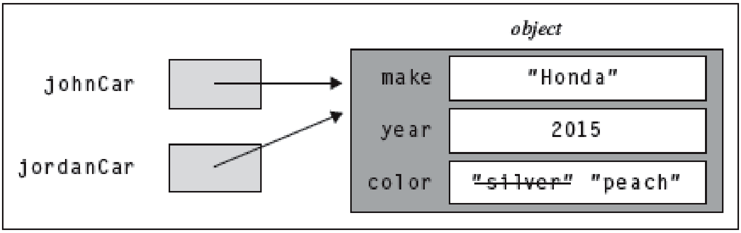
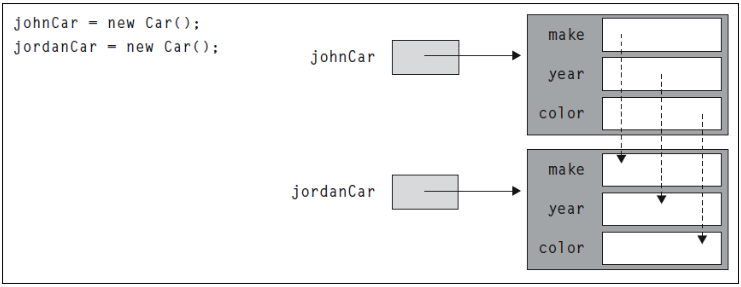

object
johnCar
jordanCar
make
"Honda"
year
2015
color
~~"silver"~~ "peach"

Figure 7.1a Effect of assignment: `jordanCar = johnCar;` Both reference variables refer to exactly the same object.

If you want to make a copy of a referenced object, you should not assign its reference to another reference. Instead, you should instantiate a new object for the second reference. Then go through each instance variable in the original object, assigning it to its corresponding instance variable in the new object. Figure 7.1b shows what we're talking about, with each dashed line representing an instance variable assignment from the original object to the new object.



```
johnCar = new Car();
jordanCar = new Car();
```

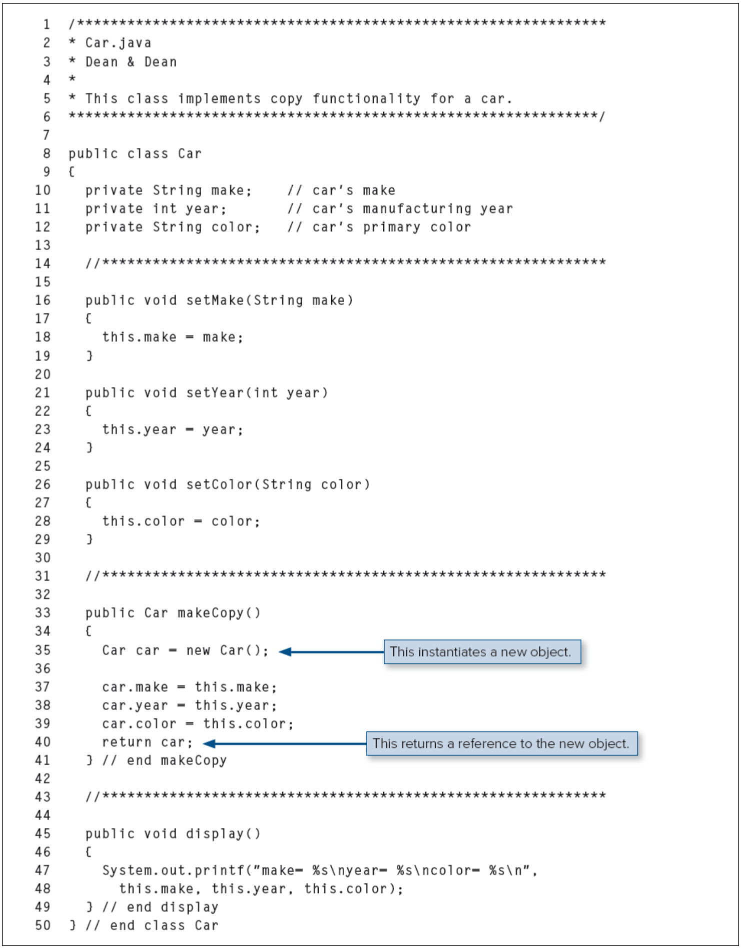
Figure 7.1b Effect of instantiating two separate objects and copying instance variable values from first object into instance variables of second object. Solid arrows are references. Dashed arrows are assignment operations.

Page 276

To illustrate the strategy outlined in Figure 7.1b, we present the Car program in Figures 7.2 and 7.3. The code includes line numbers to facilitate tracing in an end-of-chapter exercise. Look at the `makeCopy` method in the `Car` class in Figure 7.2. As its name implies, that's the method that's in charge of making a copy of a `Car` object. The `makeCopy` method instantiates a new `Car` object and assigns its reference to a local variable named `car`. Then it copies each of the calling object's instance variable values into `car`'s instance variables. Then it returns `car` to the calling module. By returning `car`, it returns a reference to the newly instantiated `Car` object.

Now look at the driver in Figure 7.3. Note how `main` assigns `makeCopy`'s returned value to `jordanCar`. After `jordanCar` gets the reference to the newly created `Car` object, it calls `setColor` to change the `Car` object's color. Because `jordanCar` and `johnCar` refer to two separate objects, the `jordanCar.setColor("peach")` method call updates only the `jordanCar` object, not the `johnCar` object. Yay!

Whenever a method finishes, its parameters and locally declared variables are deleted. In our traces, we represent this deletion by drawing a heavy line under all of the terminating method's parameters and local variables. In the `makeCopy` method in Figure 7.2, there is one local variable, the reference variable, `car`. When the `makeCopy` method finishes, the `car` reference variable is deleted. When a reference variable is deleted, the reference it holds is lost, and if that reference is not saved in a separate variable, the program will have no way of finding the object it referred to. In the `makeCopy` method, the `car` reference variable's value does get saved. The assignment is valid because it takes place before `makeCopy`'s local variable gets deleted.



```
/*********************************************************************
* Car.java
* Dean & Dean
*
* This class implements copy functionality for a car.
*********************************************************************/

public class Car
{
  private String make;    // car's make
  private int year;       // car's manufacturing year
  private String color;   // car's primary color

  //*****************************************************************

  public void setMake(String make)
  {
    this.make = make;
  }

  public void setYear(int year)
  {
    this.year = year;
  }

  public void setColor(String color)
  {
    this.color = color;
  }

  //*****************************************************************

  public Car makeCopy()
  {
    Car car = new Car();

    car.make = this.make;
    car.year = this.year;
    car.color = this.color;
    return car;
  } // end makeCopy

  //*****************************************************************

  public void display()
  {
    System.out.printf("make= %s\nyear= %s\ncolor= %s\n",
      this.make, this.year, this.color);
  } // end display
} // end class Car
```

This instantiates a new object.

This returns a reference to the new object.

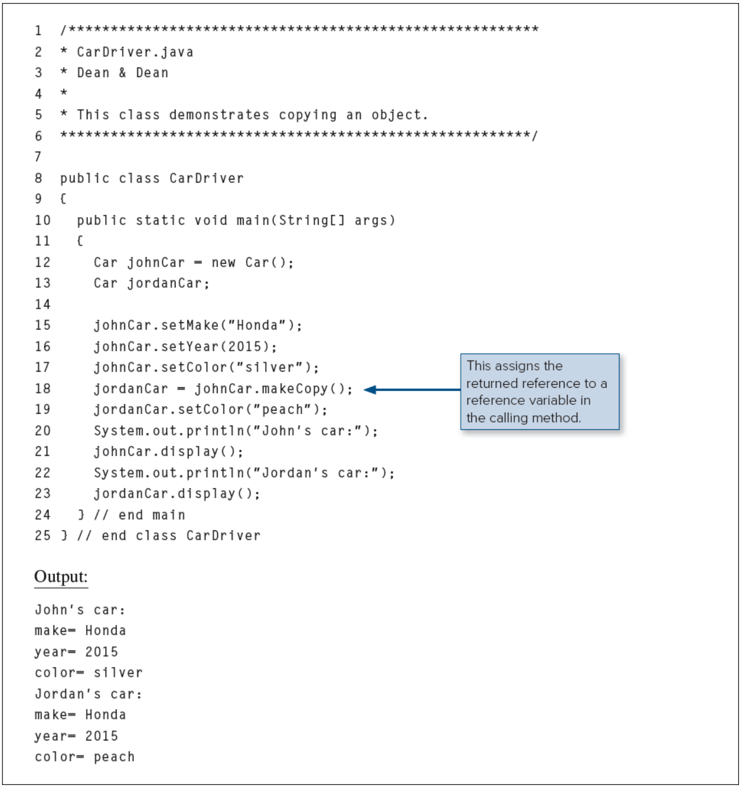
Figure 7.2 Car class with a makeCopy method that returns a reference to a copy of the calling object

Inaccessible Objects and Garbage Collection

Sometimes you'll want to instantiate a

temporary object inside a method, use it for some purpose in that method, and then abandon that object when the method finishes. At other times, you may wish to abandon an object before a method finishes. For example, suppose that in the `main` method in Figure 7.3, after calling `makeCopy` and creating a new `Car` object for `jordanCar`, you want to model John’s old car being destroyed in a fire and Jordan volunteering to let him become a co-owner of her new car. You could represent this joint ownership of one car with the statement:

```
johnCar = jordanCar;
```



```
/*************************************************************
* CarDriver.java
* Dean & Dean
*
* This class demonstrates copying an object.
*************************************************************/

public class CarDriver
{
  public static void main(String[] args)
  {
    Car johnCar = new Car();
    Car jordanCar;

    johnCar.setMake("Honda");
    johnCar.setYear(2015);
    johnCar.setColor("silver");
    jordanCar = johnCar.makeCopy();
    jordanCar.setColor("peach");
    System.out.println("John's car:");
    johnCar.display();
    System.out.println("Jordan's car:");
    jordanCar.display();
  } // end main
} // end class CarDriver
```

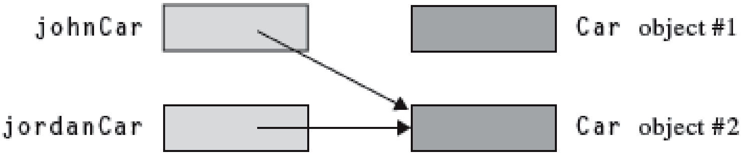
This assigns the returned reference to a reference variable in the calling method.

Output:

```
John's car:
make= Honda
year= 2015
color= silver
Jordan's car:
make= Honda
year= 2015
color= peach
```

Figure 7.3 `CarDriver` class that drives the `Car` class in Figure 7.2

Doing this overlays `johnCar`'s previous reference to John's original `Car` object, and that `Car` object - becomes inaccessible to the program (abandoned), like `Car` object #1 is in this picture:



The question is, how does the Java virtual machine (JVM) treat abandoned or inaccessible objects? Inaccessible objects can’t participate in the program, so there’s no need to keep them around. They become “garbage.” In fact, it would be bad to keep them around because they can lead to clogging up the computer’s memory. A computer has a finite amount of memory, and each piece of garbage uses up some of that memory. And that means less memory is available for new tasks. If garbage is allowed to accumulate unabated, it would eventually chew up all the *free space* in a computer’s memory (free space is the portion of memory that is unused). If there’s no free space in memory, there’s no space for any new objects, and typically the computer stops working until a reboot kicks the clutter away.

If an inaccessible object persists and uses up space in a computer's memory, that's called a *memory leak*. Memory leaks can occur in computer programs that allocate memory during execution. When a computer language requires the programmer to do something specific to prevent memory leaks, and the programmer forgets to do that, a nasty bug is born—a bug that is very hard to find. In creating the Java language, James Gosling and his Sun coworkers realized this, and they opted to make the language itself deal with the problem. How? By going into the garbage collection business. Not what Dirk and Lenny do when they pick up the trash at your curb every Tuesday, but Java *garbage collection*! Actually, James Gosling didn't invent garbage collection; it's been around since the dawn of garbage. But Java is the first popular programming language to include it as a standard service.[1]

The Java garbage collector is a utility program that searches for inaccessible objects and recycles the space that they occupy by asking the operating system to designate their space in memory as free space. This space might not be used right away, and some computer whiz kid might be able to find some of those old abandoned objects—like wandering through a trash dump, fighting off mean dogs, and looking for furniture—but for practical purposes, you should consider those abandoned objects unrecoverable and gone.

The beauty of Java's automatic garbage collection is that the programmer doesn't have to worry about it—it just happens whenever it's appropriate. And when is it appropriate? Whenever the computer is running low on free space in memory or whenever nothing else is happening, such as when a program is waiting for keyboard input. At that point, the operating system wakes up his buddy, the Java garbage collector, and tells him to go earn his keep.

7.4 Testing Objects for Equality

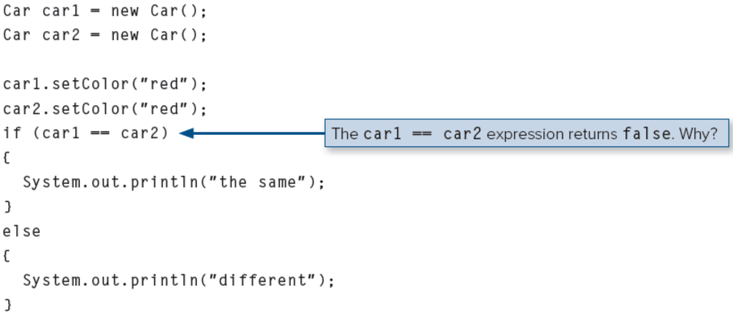
The previous section illustrated returning a reference from a method. This section illustrates passing a reference to a method to allow the method to read the referenced object's data. One of the most common applications of this occurs in testing two objects for equality. Before looking at this application, it's appropriate to look at the simplest way to evaluate equality.

The = = Operator

The `==` operator works the same for primitive variables and for reference variables. It tests if the values stored in these variables are the same. When applied to reference variables, the `==` operator returns `true` if and only if the two reference variables refer to the same object; that is, the two reference variables contain the same address and thus are aliases for the same object. For example, what does the following code fragment print?

```
Car car1 = new Car();
Car car2 = car1;
if (car1 == car2)
{
  System.out.println("the same");
}
else
{
  System.out.println("different");
}
```

It prints “the same” because `car1` and `car2` hold the same value—the address of the lone `Car` object. But if you want to see if two different objects have the same instance-variable values, the == operator is not what you want. For example, what does this code print?



```
Car car1 = new Car();
Car car2 = new Car();

car1.setColor("red");
car2.setColor("red");
if (car1 == car2)
{
  System.out.println("the same");
}
else
{
  System.out.println("different");
}
```

The `car1 == car2` expression returns `false`. Why?

This code prints “different” because `car1 == car2` returns `false`. It doesn’t matter that `car1` and `car2` contain the same data (red). The == operator doesn’t look at the object’s data; it just looks at whether the two reference variables point to the same object. In this case, `car1` and `car2` refer to distinct objects, with different storage locations in memory.

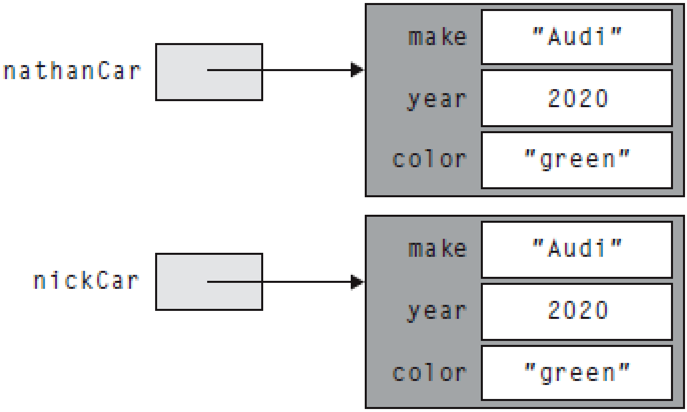
The equals Method

An `equals` method is a handy utility.

If you want to see whether two different objects have the same characteristics, you need to compare the contents of two objects rather than just whether two reference variables point to the same object. To do that, you need an `equals` method in the object's class definition that compares the two objects' instance variables. Having such an equals method is very common because you often want to test two objects to see whether they have the same characteristics. For Java's API classes, you should use the classes' built-in `equals` methods. For example, in comparing the contents of two strings, call the `String` class's `equals` method. For classes that you implement yourself, adopt the habit of writing your own `equals` methods.

An Example

The following diagram depicts two objects with identical instance variable values. Comparing `nathanCar` to `nickCar` with the `==` operator generates `false` because the two reference variables point to different objects. However, comparing `nathanCar` to `nickCar` with a standard `equals` method generates `true` because a standard `equals` method compares instance variable values, and these two objects have identical instance variable values.


nathanCar
make
"Audi"
year
2020
color
"green"
nickCar
make
"Audi"
year
2020
color
"green"

The Car2 program in Figures 7.4 and 7.5 illustrates this example. Figure 7.5's Car2 class defines an equals method, and Figure 7.4's Car2Driver class calls the equals method while comparing two Car2 objects. As is common with equals method calls, Figure 7.4's equals method call is embedded in the condition of an if statement. That should make sense when you realize that an if statement condition must evaluate to true or false and an equals method does indeed evaluate to true or false. Typically, an equals method evaluates to true if the instance variables in two objects contain the same data values, and it evaluates to false otherwise. For our Car2 program, the equals method evaluates to true if nathanCar contains the same data (make, year, and color) as nickCar. Figure 7.4 shows that -nathanCar and nickCar are assigned the same data. Therefore, the equals method returns true and the program prints "Cars have identical features."

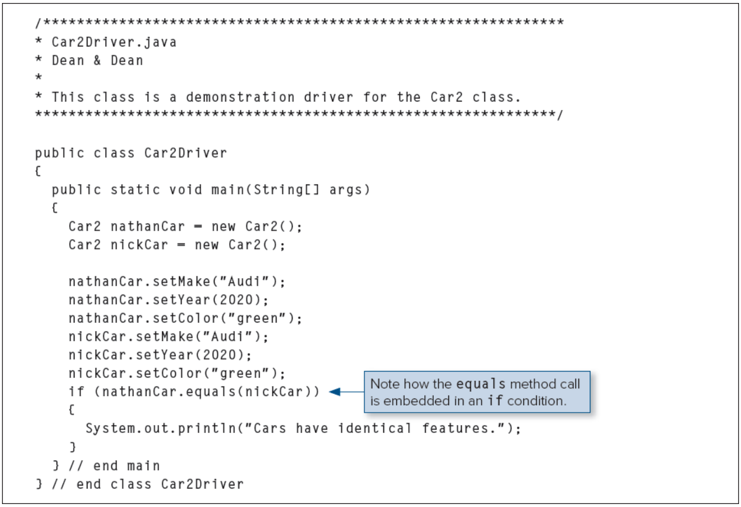
In the equals method call, note how the first Car2 reference variable, nathanCar, appears at the left of the .equals and the second Car2 reference variable, nickCar, appears inside the parentheses. Thus, nathanCar is the calling object, and nickCar is an argument. This happens a lot when using two reference variables with a method call—one reference variable will be the calling object and the other one will be the argument.

Let's now examine the equals method definition in Figure 7.5. First, note the equals method heading. Why is the return type boolean? Because the return type must match the type of the returned value, and equals methods always return a Boolean value (either true or false). Also, note that the type of the otherCar parameter is Car2. That should make sense when you look back at the equals method call in Figure 7.4. It shows that the argument being passed into the equals method is nickCar, and nickCar is a Car2 reference variable.

OK, now it’s time to examine the body of the equals method. Notice that there is just one statement—the return statement. The return value must be a boolean, so the expression after the word return must evaluate to either true or false. This expression is an “anding” together of three boolean subexpressions, each of which evaluates to either true or false. For the overall expression to be true, all three of the subexpressions must be true.

Each subexpression checks whether a particular instance variable has the same value in the calling object and the passed-in parameter object. For example, to check whether the year instance variable has the same value in the calling object and the passed-in parameter object, we do this:

```
this.year == otherCar.year
```



```
/*******************************************************************
* Car2Driver.java
* Dean & Dean
*
* This class is a demonstration driver for the Car2 class.
*******************************************************************/

public class Car2Driver
{
  public static void main(String[] args)
  {
    Car2 nathanCar = new Car2();
    Car2 nickCar = new Car2();

    nathanCar.setMake("Audi");
    nathanCar.setYear(2020);
    nathanCar.setColor("green");
    nickCar.setMake("Audi");
    nickCar.setYear(2020);
    nickCar.setColor("green");
    if (nathanCar.equals(nickCar))
    {
      System.out.println("Cars have identical features.");
    }
  } // end main
} // end class Car2Driver
```

Note how the `equals` method call is embedded in an `if` condition.

Figure 7.4 `Car2Driver` class that drives the `Car2` class in Figure 7.5

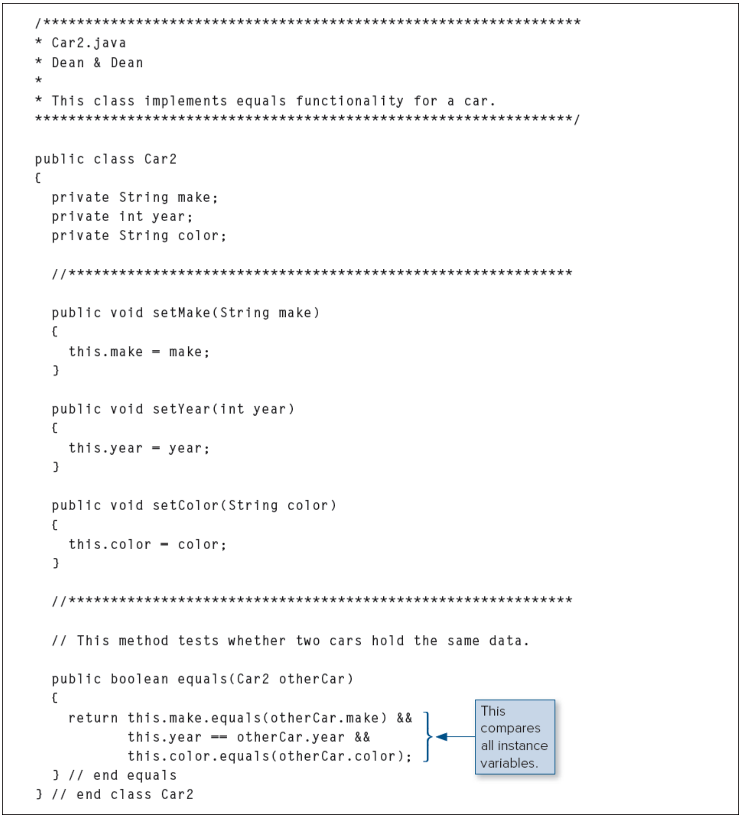
Page 282

In this case, we use the == operator to check for equality. That works fine for the year instance variable because year is an int. But the make and color instance variables are strings, and the == operator is anathema to strings. We must use the equals method for strings! Thus, to check whether the make instance variable has the same value in the calling object and the passed-in parameter object, we do this:

```
this.make.equals(otherCar.make)
```

Hmmm . . . . Does it strike you as odd to use the String class's equals method inside our Car2 class's equals method? That's perfectly OK—the compiler doesn't care if two methods happen to have the same name, as long as they are in different classes. That's part of the beauty of encapsulation!

Can you think of another way to write the body of the Car2 class's equals method? We might have used that boolean expression to the right of the return keyword as the condition of an if statement and then put return true in the if clause and return false in the else clause. But that would have been a harder and longer way to do the same thing—and probably more confusing, too, because it would have required more parentheses. Although Figure 7.5's return statement might appear at first glance to be a Cerberean rat's nest,[2] most veteran programmers would consider it to be rather elegant.



```
/****************************************************************
* Car2.java
* Dean & Dean
*
* This class implements equals functionality for a car.
****************************************************************/

public class Car2
{
  private String make;
  private int year;
  private String color;

  //************************************************************

  public void setMake(String make)
  {
    this.make = make;
  }

  public void setYear(int year)
  {
    this.year = year;
  }

  public void setColor(String color)
  {
    this.color = color;
  }

  //************************************************************

  // This method tests whether two cars hold the same data.

  public boolean equals(Car2 otherCar)
  {
    return this.make.equals(otherCar.make) &&
           this.year == otherCar.year &&
           this.color.equals(otherCar.color);
  } // end equals
} // end class Car2
```

This compares all instance variables.

Figure 7.5 `Car2` class with an `equals` method

Page 283

Suppose you want uppercase colors to be considered the same as lowercase colors. In other words, you want a silver 2018 Ford to be considered the same as a silver 2018 Ford. How should you change the code to handle that? Use equalsIgnoreCase instead of equals when comparing the color strings:

```
this.color.equalsIgnoreCase(otherCar.color)
```

This shows that you can make your equals method return true when there is only approximate equality, where you define “approximate” however you wish. We’ll discuss the equals method in more depth in Chapter 14.

7.5 Passing References as Arguments

By now, you should be fairly comfortable with the concept of passing an argument to a method. We’ve covered all you need to know about passing primitive types as arguments. But you still need to know a bit more about passing references as arguments. In the example in Figure 7.4, we passed the nickCar reference as an argument to the equals method. The equals method call assigned the nickCar reference to its otherCar parameter, and then it used the otherCar parameter to read the object's data. In that example, we used a passed-in reference to read an object's data. Now let's use a passed-in reference to update an object's data.

Suppose you pass a reference variable to a method, and inside the method, you update the referenced object's instance variables. What happens? Remember that a reference variable holds the address of an object, not the object itself. So in passing a reference variable argument to a method, a copy of the object's address (not a copy of the object itself) is passed to the method and stored in the method's parameter. Because the parameter and the argument hold the same address value, they point to the same object. Thus, if the parameter's instance variables are updated, then the update simultaneously updates the argument's instance variables in the calling module. This is a case where aliasing (using two names for the same thing) is really handy.

Person-Swapping Example

Let's see if you understand all of this reference-passing stuff by putting it in the context of a complete program. See the Person program in Figures 7.6 and 7.7. The Person program swaps names for two Person objects. As shown in Figure 7.6's main method, the person1 reference variable starts with the name "Jonathan" and the person2 reference variable starts with the name "Benji." After the swapPerson method call, person1 has the name "Benji," and person2 has the name "Jonathan." The swapPerson method swaps the names by taking advantage of the phenomenon discussed above—if a reference variable is passed to a method, then the parameter and the argument refer to the same object, and an update to one means an update to the other as well. Bottom line: When you pass a reference to a method, you enable the method to modify the referenced object.

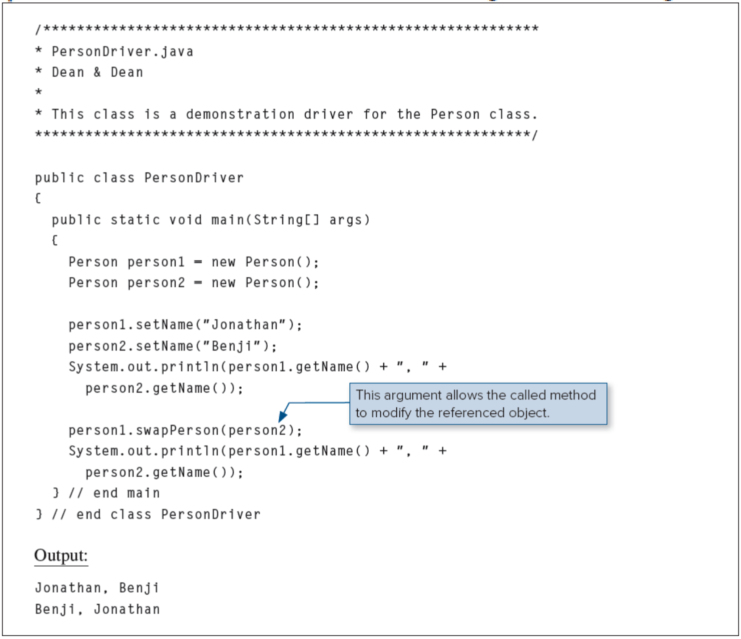
General-Purpose Swapping Algorithm

How do you swap two values?

Before digging deeper into the Person program's code, let's come up with a general-purpose swapping algorithm. Having to swap two values is a very common programming requirement, so you should make sure that you fully understand how to do it.

Suppose you're asked to provide an algorithm that swaps the contents of two variables, *x* and *y*. To make the goal more concrete, you are given the following algorithm skeleton. Replace *<insert swap code here>* with appropriate pseudocode so that the algorithm prints x = 8, y = 3.

```
x ← 3
y ← 8
<insert swap code here>
print "x = " + x + ", y = " + y
```



```
/*************************************************************
* PersonDriver.java
* Dean & Dean
*
* This class is a demonstration driver for the Person class.
*************************************************************/

public class PersonDriver
{
  public static void main(String[] args)
  {
    Person person1 = new Person();
    Person person2 = new Person();

    person1.setName("Jonathan");
    person2.setName("Benji");
    System.out.println(person1.getName() + ", " +
      person2.getName());

    person1.swapPerson(person2);
    System.out.println(person1.getName() + ", " +
      person2.getName());
  } // end main
} // end class PersonDriver
```

This argument allows the called method to modify the referenced object.

Output:

```
Jonathan, Benji
Benji, Jonathan
```

Figure 7.6 Driver for a program that implements swapping by passing a reference to a method

Would the following code work? Would it swap *x* and *y*’s contents successfully?

```
y ← x
x ← y
```

The first statement puts x’s original value into *y*. The second statement attempts to put *y*’s original value into *x*. Unfortunately, the second statement doesn’t work because *y*’s original value is gone (overwritten by *x* in the first statement). If you inserted the above code into the above algorithm, the algorithm would print:

```
x = 3, y = 3
```

Swapping requires a temporary variable.

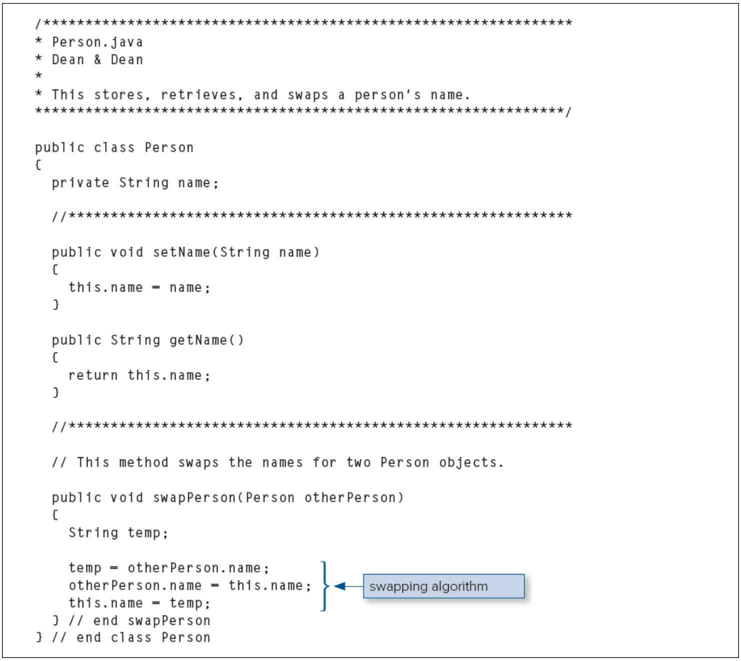
That’s not what you want! The trick is to save the value of y before you wipe it out with x’s value. How do you save it? Use a temporary variable like this:

```
temp ← y
y ← x
x ← temp
```

Person-Swapping Example—

Continued

Now look at the Person class in Figure 7.7. In particular, let’s examine how the swapPerson method implements the swapping algorithm. The swapped items are the passed-in object’s name and the calling object’s name. The passed-in object is accessed via the otherPerson parameter. Note how we access the passed-in object’s name with otherPerson.name. And note how we access the calling object’s name with this.name. And finally, note how we use a temp local variable as temporary storage for otherPerson.name.



```
/***************************************************************
* Person.java
* Dean & Dean
*
* This stores, retrieves, and swaps a person's name.
***************************************************************/

public class Person
{
  private String name;

  //************************************************************

  public void setName(String name)
  {
    this.name = name;
  }

  public String getName()
  {
    return this.name;
  }

  //************************************************************

  // This method swaps the names for two Person objects.

  public void swapPerson(Person otherPerson)
  {
    String temp;

    temp = otherPerson.name;
    otherPerson.name = this.name;
    this.name = temp;
  } // end swapPerson
} // end class Person
```

Figure 7.7 `Person` class, which implements swapping by passing a reference to a method

7.6 Method-Call Chaining

At this point, you should be fairly comfortable with calling a method. Now it's time to go one step further. In this section, you will learn to call several methods in succession, all within one statement. That's called *method-call chaining,* and it can lead to more compact code.

If you look back at Figures 7.3 and 7.4, you'll see several instances where we call several methods one after another, and we use a separate statement for each successive method call, like this code fragment from Figure 7.4:

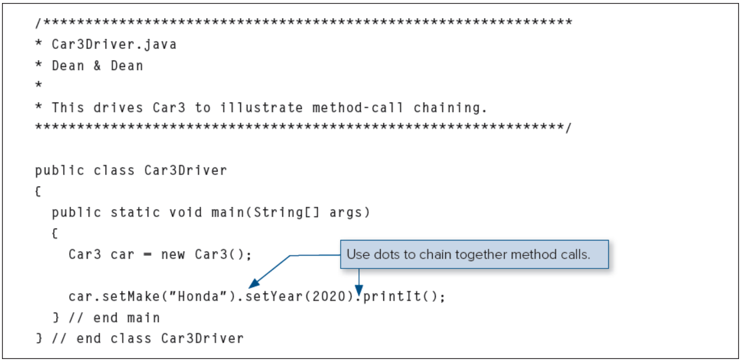
```
nathanCar.setMake("Audi");
nathanCar.setYear(2020);
```

Wouldn't it be nice to be able to chain the method calls together like this?

```
nathanCar.setMake("Audi").setYear(2020);
```

Method-call chaining is an option, not a requirement. So why use it? Because it can often lead to more elegant code—more compact and easier to understand. But don't overdo it. Using just the right amount of chaining is an art, and moderation is a virtue.

Let's look at method-call chaining in the context of a complete program. See the method-call chain (indicated by a callout) in Figure 7.8's Car3Driver class. Because left-to-right precedence applies, the car.setMake method executes first. The setMake method returns the calling object, which is the car object at the left of car.setMake. The returned car object is then used to call the setYear method. In a similar fashion, the setYear method returns the calling object, car, and it is used to call the prinIt method.



```
/****************************************************************
* Car3Driver.java
* Dean & Dean
*
* This drives Car3 to illustrate method-call chaining.
****************************************************************/

public class Car3Driver
{
  public static void main(String[] args)
  {
    Car3 car = new Car3();

    car.setMake("Honda").setYear(2020).printIt();
  } // end main
} // end class Car3Driver
```

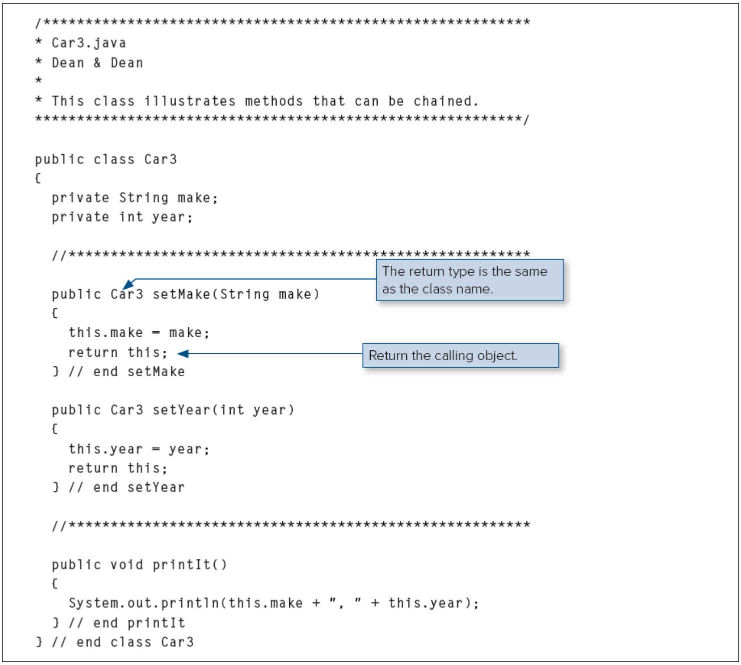
Use dots to chain together method calls.

Figure 7.8 Car3 program driver, which illustrates method-call chaining

Method call chaining doesn't work by default. If you want to enable method-call chaining for methods from the same class, you need the following two items in each method definition:

1. The last line in the method body should return the calling object by specifying return this;
2. In the method heading, the return type should be the method's class name.

We've implemented those items in the Car3 class in Figure 7.9. Verify that setMake and setYear are enabled properly for method-call chaining. Specifically, verify that (1) the last line in each method body is return this;, and (2) in each method heading, the return type is the method's class name, Car3.



```
/*************************************************************
* Car3.java
* Dean & Dean
*
* This class illustrates methods that can be chained.
*************************************************************/

public class Car3
{
  private String make;
  private int year;

  //***********************************************************

  public Car3 setMake(String make)
  {
    this.make = make;
    return this;
  } // end setMake

  public Car3 setYear(int year)
  {
    this.year = year;
    return this;
  } // end setYear

  //***********************************************************

  public void printIt()
  {
    System.out.println(this.make + ", " + this.year);
  } // end printIt
} // end class Car3
```

The return type is the same as the class name.

Return the calling object.

Figure 7.9 Car3 class

Page 288

Whenever you finish a method with a return this; statement, you're making it possible to use the same object to call the next method in the chain. However, you can also chain methods called by different types of objects. Just arrange the chain so that the reference type returned by each preceding method matches the class of each following method. So, in general, to make a method chainable, do these two things:

1. In the method heading, specify the return type as the class of a potential following method.
2. Finish the method body with:

```
return <reference to object that will call the following method>;
```

Here is a code fragment from earlier in the book that illustrates chaining of two methods defined in the Java API:

```
ch = stdIn.nextLine().charAt(0);
```

The stdIn variable is a reference to an object of the Scanner class. It calls Scanner's nextLine method, which returns a reference to an object of the String class. Then that object calls String's charAt method, which returns a character.

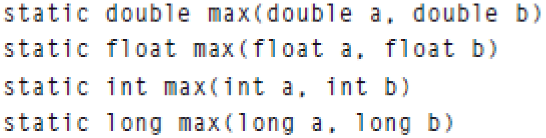
7.7 Overloaded Methods

Up until this point, all of the methods we defined for a given class have had unique names. But if you think back to some of the Java API methods presented in Chapter 5, you'll recall that there were several examples where the same name identified more than one method in the same class (For example, there are two Math.abs methods—one for a double and one for an int). This section will show you how to do this in classes you write.

What Are Overloaded Methods?

Overloaded methods are two or more methods in the same class that use the same name to indicate that they perform the same kind of operation in a different context. Since they use the same name, the compiler needs something else besides the name in order to distinguish them. Parameters to the rescue! The parametersestablish the context. To make two overloaded methods distinguishable, you define them with different parameters. More specifically, you define them with a different number of parameters or different types of parameters. The combination of a method's name, the number of its parameters, and the types of its parameters is called the method's *signature*. Each distinct method has a distinct signature.

The Java API library provides many examples of overloaded methods. Here are the headings of four overloaded max methods from the Math class:



```
static double max(double a, double b)
static float max(float a, float b)
static int max(int a, int b)
static long max(long a, long b)
```

Note that each method heading has a distinct signature because each heading is distinguishable in terms of number and types of parameters.

Suppose you're writing software for an alarm system that uses a word, a number, or a combination of a word and a number for the alarm system's password. To activate (or deactivate) the alarm, your program first needs to call a validation method that checks the passed-in word, number, or a combination of word and number. Here are the headings for overloaded methods that could implement that functionality:

```
boolean validate(String word)
boolean validate(int number)
boolean validate(String word, int num)
boolean validate(int num, String word)
```

Here’s how you could call the first and second methods:

```
if (alarm.validate("soccer"))
{
  ...
}
else if (alarm.validate(54321))
{
  ...
```

The previous code works fine, but suppose you try to implement an additional validate method to handle a different kind of password—one that allows interspersed letters and digits within one string. Here is the proposed method heading:

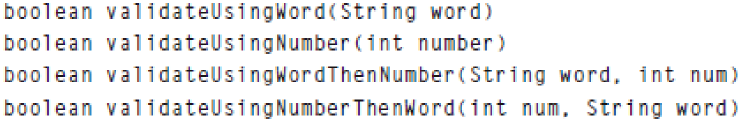
```
void validate(String code)
```

Would such a method be legal? No. Its signature is the same as the first validate method's signature—same method names and same number and types of parameters. It doesn't matter that the parameter names are different—word and code. It's the parameter types that matter, not the parameter names. It doesn't help that the return types are different—boolean and void. The return type is not part of the signature. Because the signatures are the same but the return types are different, if you try to include these two method headings in the same class, the compiler will think you're trying to define two different versions of exactly the same method. That will irritate the compiler. Be prepared for it to snarl back at you with a "duplicate definition" compile-time error message.

Benefit of Overloaded Methods

When should you use overloaded methods? When you need to perform essentially the same task with different parameters. For example, the validate methods described above perform essentially the same basic task—they determine whether a given value or values form a valid password. But they perform the task on different sets of parameters. Given that situation, overloaded methods are a perfect fit.

Note that the use of overloaded methods is never an absolute requirement. As an alternative, you can use different method names to distinguish different methods. So why are the above validate method headings better than the below method headings?



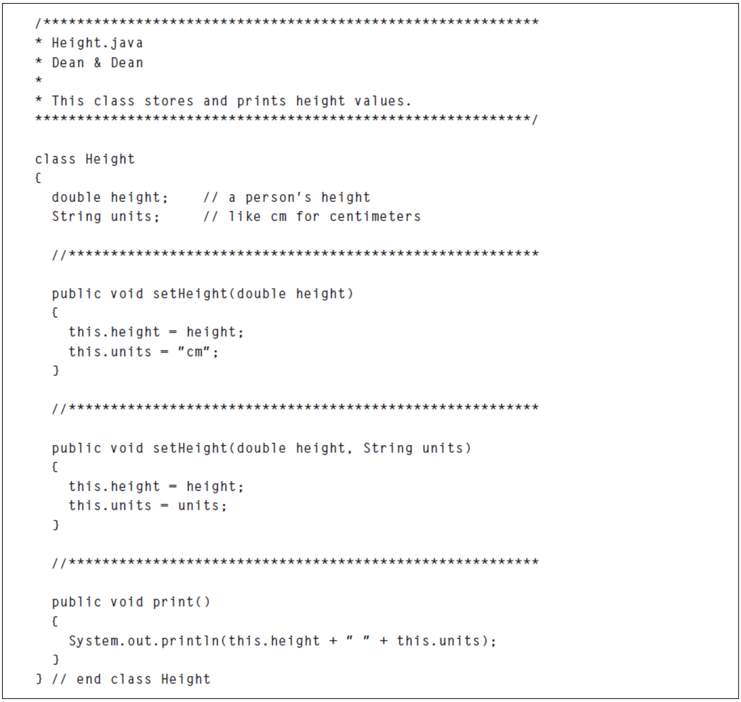
```
boolean validateUsingWord(String word)
boolean validateUsingNumber(int number)
boolean validateUsingWordThenNumber(String word, int num)
boolean validateUsingNumberThenWord(int num, String word)
```

As these examples suggest, using different method names is cumbersome. With only one method name, the name can be simple. As a programmer, wouldn't you prefer to use and remember just one simple name rather than several cumbersome names?

A Complete Example

Look at the class in Figure 7.10. It uses overloaded setHeight methods. Both methods assign a height parameter to a height instance variable. The difference is the technique for assigning the height's units. The first method automatically assigns a hard-coded "cm" (for centimeters) to the units instance variable. The second method assigns a user-specified units parameter to the units instance variable. The second method thus requires two parameters, height and units, whereas the first method requires only one parameter, height. The two methods perform pretty much the same task, with only a slight variation. That’s why we want to use the same name and “overload” that name.

Now look at the driver in Figure 7.11 and its two setHeight method calls. For each method call, can you tell which of the two overloaded methods is called? Figure 7.11’s first method call, setHeight(72.0, "in"), calls Figure 7.10’s second setHeight method because the two arguments in the method call match the two parameters in the second method’s heading. Figure 7.11’s second method call, setHeight(180.0), calls Figure 7.10’s first setHeight method because the one argument in the method call matches the one parameter in the first method’s heading.



```
/*************************************************************
* Height.java
* Dean & Dean
*
* This class stores and prints height values.
*************************************************************/

class Height
{
  double height;    // a person's height
  String units;     // like cm for centimeters

  //**********************************************************

  public void setHeight(double height)
  {
    this.height = height;
    this.units = "cm";
  }

  //**********************************************************

  public void setHeight(double height, String units)
  {
    this.height = height;
    this.units = units;
  }

  //**********************************************************

  public void print()
  {
    System.out.println(this.height + " " + this.units);
  }
} // end class Height
```

Figure 7.10 `Height` class with overloaded methods

Calling an Overloaded Method from Within an Overloaded Method

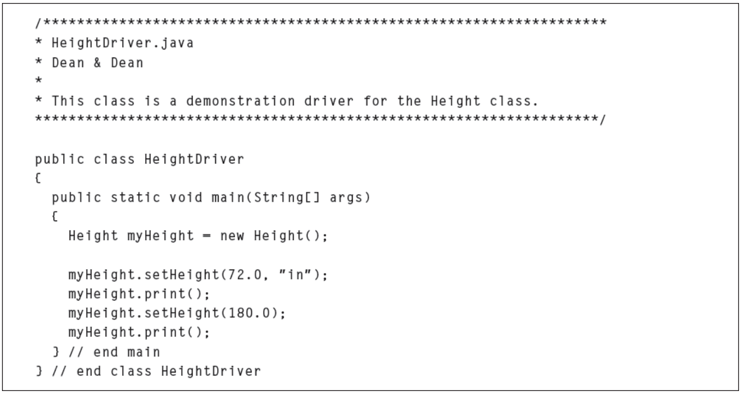
Suppose you have overloaded methods and you want one of the overloaded methods to call another one of the overloaded methods. Figure 7.12

provides an example that shows how to do that. Figure 7.12's setHeight method is an alternative version of Figure 7.10's one-parameter setHeight method. Note how it calls the two-parameter setHeight method.

The additional method call makes the program slightly less efficient, but some might consider it more elegant because it eliminates code redundancy. In Figure 7.10, this.height = height; appears in both methods, and that's code redundancy—albeit trivial code redundancy.

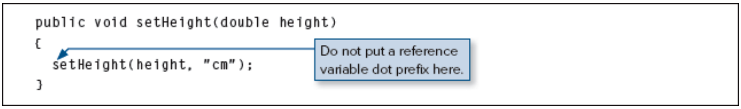
Why is there no reference variable dot at the left of the setHeight method call in the body of the method in Figure 7.12? Because if you're in an instance method, and if you call another method that's in the same class, the reference variable dot prefix is unnecessary. And in this case, the two overloaded setHeight methods are instance methods and they are indeed in the same class.

Page 292



```
/*******************************************************************
* HeightDriver.java
* Dean & Dean
*
* This class is a demonstration driver for the Height class.
*******************************************************************/

public class HeightDriver
{
  public static void main(String[] args)
  {
    Height myHeight = new Height();

    myHeight.setHeight(72.0, "in");
    myHeight.print();
    myHeight.setHeight(180.0);
    myHeight.print();
  } // end main
} // end class HeightDriver
```

Figure 7.11 `HeightDriver` class that drives the `Height` class in Figure 7.10



```
public void setHeight(double height)
{
  setHeight(height, "cm");
}
```

Do not put a reference variable dot prefix here.

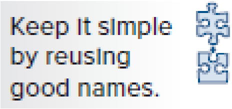
Figure 7.12 Example of a method that calls another method in the same class This helps avoid duplication of code details and possible internal inconsistencies.

With no reference variable dot prefix in Figure 7.12’s setHeight(height, "cm"); method call, you might be thinking that the method call has no calling object. Actually, there is an implied calling object; it’s the same calling object that called the current method. Review quiz: How can you access the current method's calling object? Use the this reference. If you want to make the this reference explicit, you can add it to Figure 7.12's setHeight method call as follows:

```
this.setHeight(height, "cm");
```

We point out this alternative syntax not because we want you to use it, but because we want you to get a clearer picture of calling object details.

Program Evolution


Keep it simple
by reusing
good names.

The ability to overload a method name promotes graceful program evolution because it corresponds to how natural language regularly overloads the meanings of words. For example, the first version of your program might define just the one-parameter version of its setHeight method. Later, when you decide to enhance your program, it's easier for your existing users if you minimize the new things they have to learn. In this case, you let them either keep using the original method or switch to the improved method. When they want to use the improved method, all they have to remember is the original method name and adding a second argument, for units, to the method call. That's an almost obvious variation, and it's easier to remember than a different method name. It's certainly easier than being forced to learn a new method name for the old task—which would be a necessary cost of upgrading if method overloading were not available.

7.8 Constructors

Up to this point, we have used mutators to assign values to the instance variables in newly instantiated objects. That works OK, but it requires having and calling one mutator for each instance variable. As an alternative, you could use a single method to initialize all of an object's instance variables as soon as possible after you create that object. For example, in this chapter's Car class in Figure 7.2, instead of defining three mutator methods, you could define a single initCar method to initialize Car objects. Then you could use it like this:

```
Car lanceCar = new Car();
lanceCar.initCar("Ford", 2019, "lime");
```

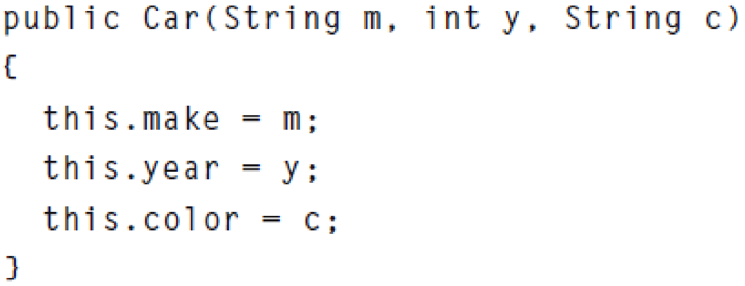
This code fragment uses one statement to allocate space for a new object, and it uses another statement to initialize that object's instance variables. Because the instantiation and initialization of an object is so common, wouldn't it be nice if there were a single statement that could handle both of these operations? There is such a statement, and here it is:

```
Car lanceCar = new Car("Ford", 2019, "lime");
```

This unifies the creation of an object and the initialization of its instance variables in just one call. It guarantees that an object's instance variables are initialized as soon as the object is created. The code that follows the word new should remind you of a method call. Both that code and a method call consist of a programmer-defined word (Car in this case) and then parentheses around a list of items. You can think of that code as a special method call, but it's so special that it has its own name. It's used to construct objects, so it's called a *constructor*.

What Is a Constructor?

A constructor is a method-like entity that's called automatically when an object is instantiated. The above new Car("Ford", 2019, "lime") object instantiation calls a constructor named Car that has three parameters—a String, an int, and a String. Here's an example of such a constructor:



```
public Car(String m, int y, String c)
{
  this.make = m;
  this.year = y;
  this.color = c;
}
```

As you can see, this constructor simply assigns passed-in parameter values to their corresponding instance variables. After executing a constructor, the JVM returns the address of the newly instantiated and initialized object to where the constructor was called. In the above Car lanceCar = new Car("Ford", 2019, "lime") declaration, the address of the instantiated Car object gets assigned to the lanceCar reference variable.

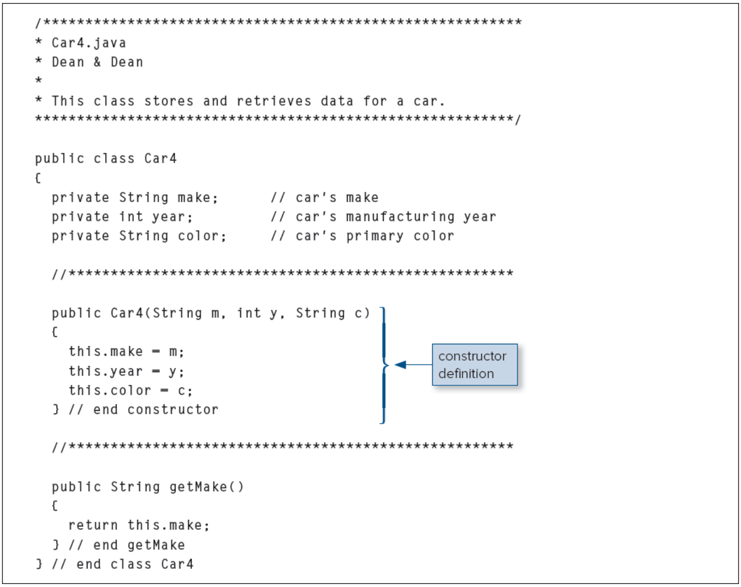
There are several constructor details you should know before looking at a complete program example. A constructor's name must be the same as the class it's associated with. Thus, a Car class's constructor must be named Car, with an uppercase "C."

In the heading of a method, you must include a return type, so you might expect the same requirement for the heading of a constructor. Nope. Return types are not used in constructor headings3 because a constructor call (with new) automatically returns a reference to the object it constructs, and the type of this object is always specified by the constructor name itself. Just specify public at the left and then write the class name (which is the name of the constructor).

A Complete Example

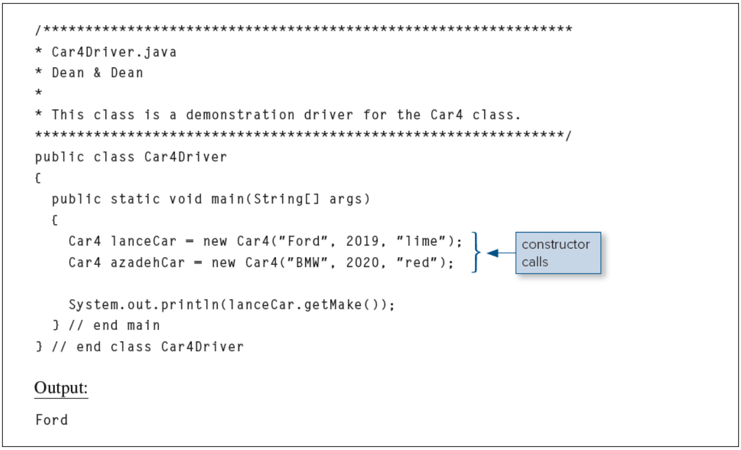
Let's now look at a complete program example that uses a constructor. See the Car4 program in Figures 7.13 and

7.14. In Figure 7.13, note that we put the constructor above the getMake method. In all class definitions, it's good style to put constructors above methods.



```
/*************************************************************
* Car4.java
* Dean & Dean
*
* This class stores and retrieves data for a car.
*************************************************************/

public class Car4
{
  private String make;      // car's make
  private int year;         // car's manufacturing year
  private String color;     // car's primary color

  //*******************************************************

  public Car4(String m, int y, String c)
  {
    this.make = m;
    this.year = y;
    this.color = c;
  } // end constructor

  //*******************************************************

  public String getMake()
  {
    return this.make;
  } // end getMake
} // end class Car4
```

Figure 7.13 `Car4` class, which has a constructor



```
/***************************************************************
* Car4Driver.java
* Dean & Dean
*
* This class is a demonstration driver for the Car4 class.
***************************************************************/
public class Car4Driver
{
  public static void main(String[] args)
  {
    Car4 lanceCar = new Car4("Ford", 2019, "lime");
    Car4 azadehCar = new Car4("BMW", 2020, "red");

    System.out.println(lanceCar.getMake());
  } // end main
} // end class Car4Driver
```

constructor calls

Output:

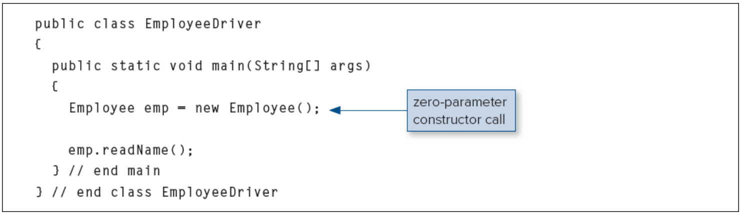
```
Ford
```

Figure 7.14 `Car4Driver` class, which drives the `Car4` class in Figure 7.13

Accommodating Java's Fickle Default Constructor

Any time you instantiate an object (with new), there must be a matching constructor. That is, the number and types of arguments in your constructor call must match the number and types of parameters in a defined constructor. But until recently, we've instantiated objects without any explicit constructor. So were those examples wrong? No. They all used a zero-parameter freebie *default constructor* that the Java compiler automatically provides if and only if there is no explicitly defined constructor. The Employee program in Figures 7.15a and 7.15b illustrates the use of Java's implicit zero-parameter default constructor.

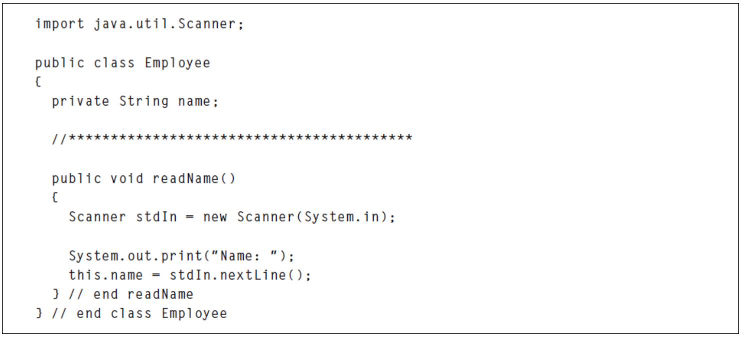
In Figure 7.15a, note how main's new Employee() code calls a zero-parameter constructor. But Figure 7.15b does not define a zero-parameter constructor. No problem. Because there are no other constructors, the Java compiler provides the default zero-parameter constructor, and it matches up with the new Employee() zero-argument constructor call.



```
public class EmployeeDriver
{
  public static void main(String[] args)
  {
    Employee emp = new Employee();

    emp.readName();
  } // end main
} // end class EmployeeDriver
```

zero-parameter constructor call

Figure 7.15a Driver for the Employee program



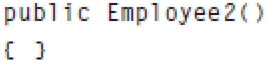
```
import java.util.Scanner;

public class Employee
{
  private String name;

  //*****************************************

  public void readName()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print(”Name: ”);
    this.name = stdIn.nextLine();
  } // end readName
} // end class Employee
```

Figure 7.15b Driven class for the Employee program This works even though there is no explicitly defined constructor because the Java compiler supplies a matching default zero-parameter constructor.

Page 296

Note that as soon as you define any kind of constructor for a class, Java's default constructor becomes unavailable. So if your class contains an explicit constructor definition, and if main includes a zero-argument constructor call, you must also include an explicit zero-parameter constructor in your class definition.

⚠ See the Employee2 program in Figures 7.16a and 7.16b. The driven class in Figure 7.16a compiles - successfully, but the driver in Figure 7.16b generates a compilation error. As in Figure 7.15a, the driver code in Figure 7.16b calls a zero-parameter constructor. It worked before, so why doesn't it work this time? This time, the driven class in Figure 7.16a explicitly defines a constructor, so Java does not provide a default zero-parameter constructor. And without that constructor, the compiler complains that there's no matching constructor for the zero-parameter constructor call. How can you fix the Employee2 program to get rid of this error? Add the following zero-parameter Employee2 constructor to your Employee2 class:



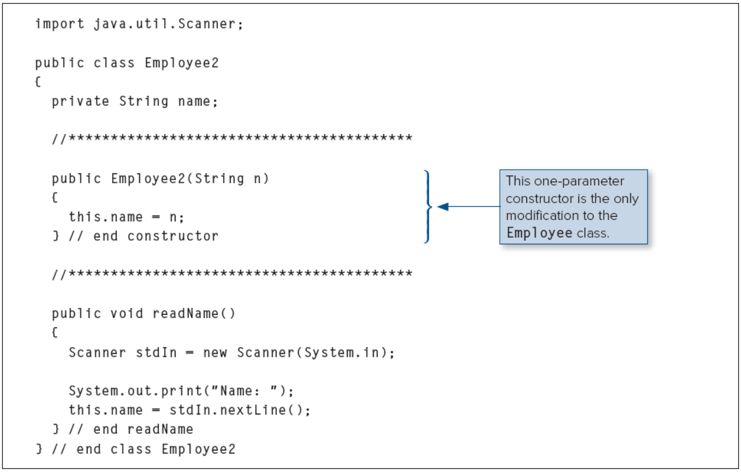
```
public Employee2()
{ }
```

That's an example of a *dummy constructor*. It's called a dummy constructor because it doesn't do anything other than satisfy the compiler. Note how the braces are on a line by themselves with a blank space between them. That's a style issue. By writing the dummy constructor like that, it makes the empty braces more prominent and clearly shows the intent of the programmer to make the constructor a dummy constructor.

Initializing Instance Constants with a Constructor

As you've seen, the purpose of a constructor is to initialize; that is, assign initial values to an object's attributes. Normally when you think of an object's attributes, you think of its instance variables. But objects can also have instance named constants as attributes (normally called *instance constants*). Constructors are especially important for initializing an object's instance constants. In this subsection, we'll discuss instance constants and how they're initialized within constructors.

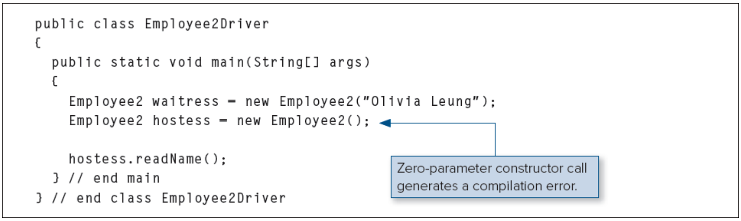
In the past, you've seen named constants declared within methods. When a named constant is declared within a method, it's called a *local named constant*, and its scope is limited to that one method. If you want an attribute that's constant throughout the life of a particular object, you'll need another kind of named constant, an instance constant. Study Figure 7.17a's Employee3 program. It improves upon our previous employee programs because it uses an instance constant to store an employee's name rather than an instance variable. Here is the instance constant declaration for an employee's name:



```
import java.util.Scanner;

public class Employee2
{
  private String name;

  //*****************************************

  public Employee2(String n)
  {
    this.name = n;
  } // end constructor

  //*****************************************

  public void readName()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Name: ");
    this.name = stdIn.nextLine();
  } // end readName
} // end class Employee2
```

This one-parameter constructor is the only modification to the `Employee` class.

Figure 7.16a Driven class for the Employee2 program



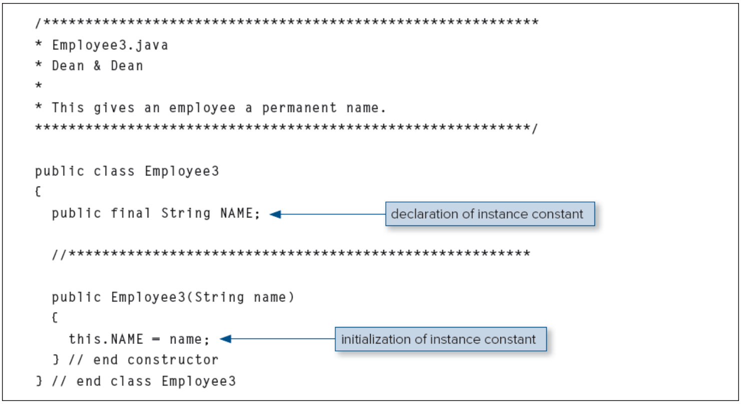
```
public class Employee2Driver
{
  public static void main(String[] args)
  {
    Employee2 waitress = new Employee2("Olivia Leung");
    Employee2 hostess = new Employee2();

    hostess.readName();
  } // end main
} // end class Employee2Driver
```

Zero-parameter constructor call generates a compilation error.

Figure 7.16b Driver for the Employee2 program

```
public final String NAME;
```

The final modifier establishes the NAME variable as a named constant. It tells the compiler to generate an error if your program attempts to assign a value to the named constant after it's been initialized. We use all uppercase for NAME because that is the standard coding convention for named constants.



```
/*************************************************************
* Employee3.java
* Dean & Dean
*
* This gives an employee a permanent name.
*************************************************************/

public class Employee3
{
  public final String NAME;

  //*********************************************************

  public Employee3(String name)
  {
    this.NAME = name;
  } // end constructor
} // end class Employee3
```

declaration of instance constant
initialization of instance constant

Figure 7.17a `Employee3` class uses an instance constant

Page 298

You should declare an instance constant at the top of its class, above the class's instance variables. Although it's legal to initialize an instance constant as part of its declaration, normally you should not do that. Instead, you should initialize it within a constructor. This allows you to initialize instance constants with different values for different objects. Thus, an instance constant can represent an attribute whose value varies from one object to another, but remains constant throughout the life of any particular object. It represents an inalienable attribute of that object, an attribute that permanently distinguishes that object from all other objects in the same class. With the Employee3 class using an instance constant for employee names, the NAME attribute reflects the fixed nature of people's names.

Because the final modifier keeps a named constant from being changed after it's initialized, it's safe to make an instance constant public. This makes it especially easy to determine the value of an object's permanent attributes. Just use this syntax:

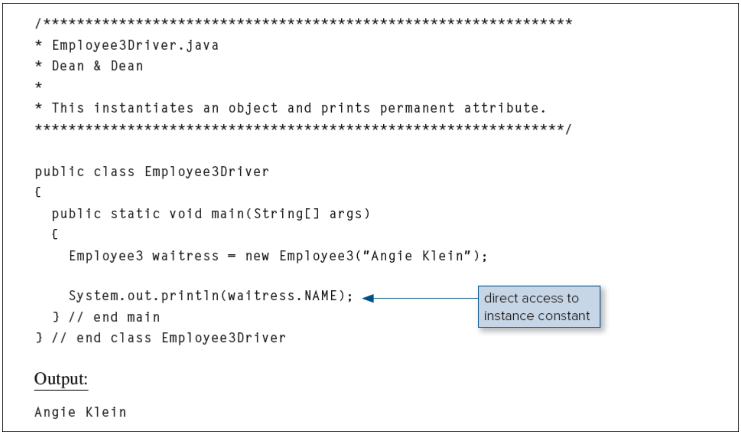
> *reference-variable.instance-constant*

In Figure 7.17a, here's how the Employee3 program's constructor uses the above syntax to initialize an employee's name (this is the reference variable for the constructor's calling object and NAME is the instance constant):

```
this.NAME = name;
```

And in Figure 7.17b, here's how the Employee3 program's main method uses the above syntax to print the waitress employee's name (waitress is the reference variable and NAME is the instance constant):

```
System.out.println(waitress.NAME);
```



```
/*****************************************************************
* Employee3Driver.java
* Dean & Dean
*
* This instantiates an object and prints permanent attribute.
*****************************************************************/

public class Employee3Driver
{
  public static void main(String[] args)
  {
    Employee3 waitress = new Employee3("Angie Klein");

    System.out.println(waitress.NAME);
  } // end main
} // end class Employee3Driver
```

direct access to instance constant

Output:

```
Angie Klein
```

Figure 7.17b Driver for the `Employee3` class in Figure 7.17a

Elegance

As described at the beginning of this section, you don’t have to implement a constructor to assign initial values to a newly created object. As an alternative, you can instantiate an object by calling the default constructor with no arguments, and then call a method that assigns initial values to the object. For example:

```
Car lexiCar = new Car();
lexiCar.initCar("Tesla", 2019, "zircon blue");
```

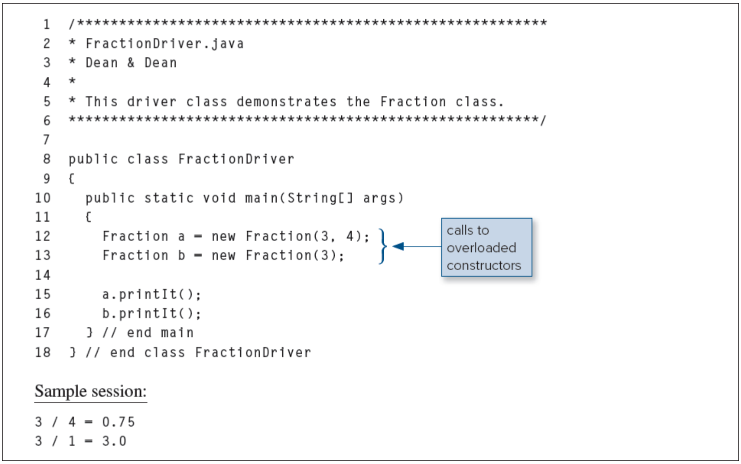
That works, but it’s much less elegant than using a constructor. By using a constructor, you intimately tie your instance variable initializations with the object you’re creating. Also, you simplify things by (1) avoiding a separate method call step and (2) avoiding the need to dream up a separate name for the initialization method (because the constructor’s name equals the class name). Bravo, constructors!

7.9 Overloaded Constructors

Overloading a constructor is like overloading a method. Constructor overloading occurs when there are two or more constructors with the same name and different parameters. Like overloaded methods, overloaded constructors have approximately the same semantics, but different syntax. Overloaded constructors are very common (more common than overloaded methods). That's because you'll often want to be able to create objects with different amounts of initialization. Sometimes you'll want to pass in initial values to the constructor. At other times, you'll want to refrain from passing in initial values to the constructor and rely on assigning values later on. To enable both of those scenarios, you need overloaded constructors—one constructor with parameters and one constructor without parameters.

An Example

Suppose you want to implement a Fraction class, which stores the numerator and denominator for a given fraction. The Fraction class also stores the fraction's quotient, which is produced by dividing the numerator by the denominator. Normally, you'll instantiate the Fraction class by passing a numerator argument and a denominator argument to a two-parameter Fraction constructor. But for a whole number, you should instantiate a Fraction class by passing just one argument (the whole number) to a Fraction constructor, rather than passing two arguments. For example, to instantiate a Fraction object for the whole number, 3, you should pass in just a 3 to a Fraction constructor, rather than a 3 for the numerator and a 1 for the denominator. To handle two-argument Fraction instantiations as well as one-argument Fraction instantiations, you need overloaded constructors. One way to begin solving a problem is to write a driver that shows how you want the solution to be used. With that in mind, we present a driver in Figure 7.18 that illustrates how the proposed Fraction class and its overloaded constructors can be used. The driver's code includes line numbers to facilitate later tracing.



```
 1  /*************************************************************
 2  * FractionDriver.java
 3  * Dean & Dean
 4  *
 5  * This driver class demonstrates the Fraction class.
 6  *************************************************************/
 7
 8  public class FractionDriver
 9  {
10    public static void main(String[] args)
11    {
12      Fraction a = new Fraction(3, 4);
13      Fraction b = new Fraction(3);
14
15      a.printIt();
16      b.printIt();
17    } // end main
18  } // end class FractionDriver
```

calls to overloaded constructors

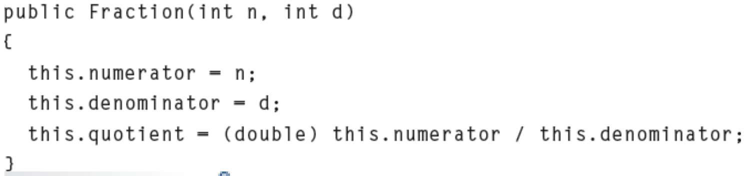
Sample session:

```
3 / 4 = 0.75
3 / 1 = 3.0
```

Figure 7.18 `FractionDriver` class,

which drives the `Fraction` class in Figure 7.19

Assume that within the Fraction class, numerator and denominator are int instance variables and quotient is a double instance variable. The two-parameter constructor should look something like this:

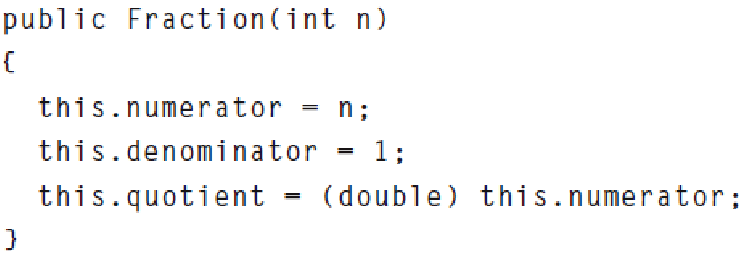


```
public Fraction(int n, int d)
{
  this.numerator = n;
  this.denominator = d;
  this.quotient = (double) this.numerator / this.denominator;
}
```

Make It robust.

Why the (double) cast? Without it, we'd get integer division and truncation of fractional values. The cast converts numerator into a double, the double numerator promotes the denominator instance variable to double, floating-point division occurs, and fractional values are preserved. Our cast to double also provides a more graceful response if the denominator is zero. Integer division by zero causes the program to crash. But floating-point division by zero is acceptable. Instead of crashing, the program prints “Infinity” if the numerator is positive or “-Infinity” if the numerator is negative.

For a whole number like 3, we could call the above two-parameter constructor with 3 as the first argument and 1 as the second argument. But we want our Fraction class to be friendlier. We want it to have another (overloaded) constructor which has just one parameter. This one-parameter constructor could look like this:



```
public Fraction(int n)
{
  this.numerator = n;
  this.denominator = 1;
  this.quotient = (double) this.numerator;
}
```

Calling a Constructor from Within Another Constructor

Avoid duplicate code.

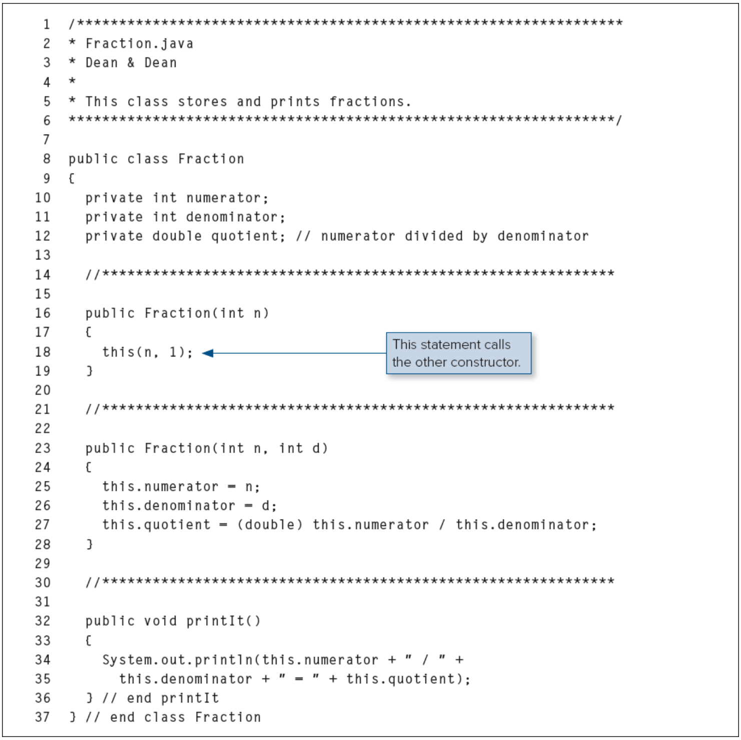
The two constructors above contain duplicate code. Duplication makes programs longer. More importantly, it introduces the possibility of inconsistency. Earlier, we used overloaded methods to avoid this kind of danger. Instead of repeating code as in Figure 7.10, in Figure 7.12, we inserted a call to a previously written method that already had the code we wanted. You do the same thing with constructors; that is, you can call a previously written constructor from within another constructor. Constructor calls are different from method calls in that they use the reserved word new, which tells the JVM to allocate space in memory for a new object. Within the original constructor, you could use the new operator to call another constructor. But that would create a separate object from the original object. And most of the time, that's not what you want. Normally, if you call an overloaded constructor, you want to work with the original object, not a new, separate object.

To avoid creating a separate object, Java designers came up with special syntax that allows an overloaded constructor to call one of its partner overloaded constructors such that the original object is used. Here is the syntax:

```
this(arguments-for-target-constructor);
```

A this(*arguments-for-target-constructor*) constructor call may appear only in a constructor definition, and it must appear as the very first statement in the constructor definition. That means you can't use this syntax to call a constructor from inside a method definition. It also means you can have only one such constructor call in a constructor definition, because only one call statement could be the "very first statement in the constructor definition."

Now look at the Fraction class in Figure 7.19. It has three instance variables—numerator, denominator, and quotient. The quotient instance variable holds the floating-point result of dividing the numerator by the denominator. The second constructor is just like the two-parameter constructor we wrote above. But the first constructor is shorter. Instead of repeating code appearing in the second constructor, it calls the second constructor with the this(...) command.



```java
/*********************************************************************
* Fraction.java
* Dean & Dean
*
* This class stores and prints fractions.
*********************************************************************/

public class Fraction
{
  private int numerator;
  private int denominator;
  private double quotient; // numerator divided by denominator

  //*****************************************************************

  public Fraction(int n)
  {
    this(n, 1);
  }

  //*****************************************************************

  public Fraction(int n, int d)
  {
    this.numerator = n;
    this.denominator = d;
    this.quotient = (double) this.numerator / this.denominator;
  }

  //*****************************************************************

  public void printIt()
  {
    System.out.println(this.numerator + " / " +
      this.denominator + " = " + this.quotient);
  } // end printIt
} // end class Fraction
```

This statement calls the other constructor.

Figure 7.19 `Fraction` class with overloaded constructors

Suppose that during program development, for debugging purposes, you decided to print “In 1-parameter constructor” from within the Fraction class’s one-parameter constructor. Where would you put that print statement? Because the this(n, 1) constructor call must be the first statement in the constructor definition, you would have to put the print statement below the constructor call.

Tracing with Constructors

Figure 7.20 shows a trace of the Fraction program. In the following discussion of it, you'll need to actively refer to not only the trace figure, but also the FractionDriver class (Figure 7.18) and the Fraction class (Figure 7.19). Note how line 12 in the FractionDriver class passes 3 and 4 to the two-parameter Fraction constructor, which assigns 3 and 4 to the constructor's n and d parameters. As part of the implied constructor functionality, lines 10–12 in the Fraction class initialize Fraction instance variables with their default values. Then lines 25–27 overwrite those initialized values. Going back to FractionDriver, new returns an object reference (obj1) to the reference variable a. Then on line 13, the driver passes 3 to the one-parameter constructor. After parameter assignment and instance variable initialization, line 18 in the Fraction class passes 3 and 1 to the two-parameter constructor. After the two-parameter constructor overwrites the instance variables, control flows back to the one-parameter constructor, and from there back to FractionDriver, where new returns an object reference (obj2) to the reference variable b. Finally, in lines 15 and 16, the driver prints out the two results.

FractionDriver			Fraction											
	main			Fraction		Fraction	printIt	obj1			obj2			
line#	a	b	*line#*	n	d	n	this	num	den	quot	num	den	quot	output
12				3	4									
			10					0						
			11						0					
			12							0.00				
			25					3						
			26						4					
			27							0.75				
12	obj1													
13						3								
			10								0			
			11									0		
			12										0.00	
			18	3	1									
			25								3			
			26									1		
			27										3.00	
13		obj2												
15							obj1							
			34											3/4=0.75
16							obj2							
			34											3/1=3.00

Figure 7.20 Trace of the Fraction program in Figures 7.18 and 7.19

7.10 Static Variables

So far in this chapter, the focus has been on the "object" part of "object-oriented programming." Thus, when you envision an object-oriented solution, you probably see separate objects, each with its own set of data and behaviors (instance variables and instance methods, respectively). That's a valid picture, but you should be aware that in addition to data and behaviors that are specific to individual objects, you also can have data and behaviors that relate to an entire class. Such data and behaviors are referred to as *static variables* and *static methods*, respectively.

Let's look at an example. Suppose

you are in charge of keeping track of YouTube videos. You need to instantiate a YouTube object for each YouTube video, and within each object, you need to store attributes like the videographer, the video's length, and the video file itself. You should store those attributes in instance variables because they are associated with individual YouTube objects. You also need to store attributes like the number of videos and the most popular video. You should store those attributes in static variables because they relate to the collection of YouTube objects as a whole.

In this section, you'll learn how to declare static variables, when to use them, what their default values are, and what their scope is. In the next section, you'll see examples of using static variables from within static methods.

Static Variable Declaration Syntax

Here is the syntax for a static variable declaration statement:

> *private-or-public* static *type variable-name*

And here is an example:

> private static int mouseCount; // total number of mouse objects

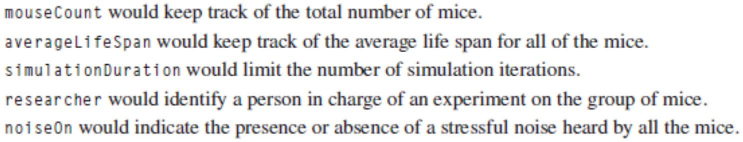
Should static variables be public or private? The philosophy on this is the same as it is for instance variables. Because you can always write public get/set static methods, you don't need public static variables any more than you need public instance variables. It's best to keep your variables as private as possible to maintain control over how they are accessed. Therefore, in addition to making instance variables - private, you should also make static variables private.

Why the Term “static”?

As you know, when the JVM sees the new operator in a program, it instantiates an object for the specified class. In so doing, it allocates memory space for all of the object’s instance variables. Later, the garbage collector might deallocate (take away) that memory space before the program stops if all references to that space disappear. That sort of memory management, done while the program runs, is called *dynamic allocation*. Static variables are different. The JVM allocates space for static variables when the program starts, and that variable space remains allocated for as long as the program runs. That sort of memory management is called static allocation. That’s why we call these variables static variables.

Static Variable Examples

As you know, each use of new creates a separate copy of all instance variables for each object. On the other hand, for a particular class, there is only one copy of each static variable, and all objects share that single copy. Thus, you should use static variables to describe properties of a class's objects that need to be shared by all of the objects. For example, consider again the problem of simulating mouse growth. In our previous mouse programs, we kept track of data pertinent to each individual mouse—a mouse's growth rate, a mouse's age, and a mouse's weight. For a more useful simulation program, you'd probably also want to keep track of group data and common environmental data. For example:



`mouseCount` would keep track of the total number of mice.

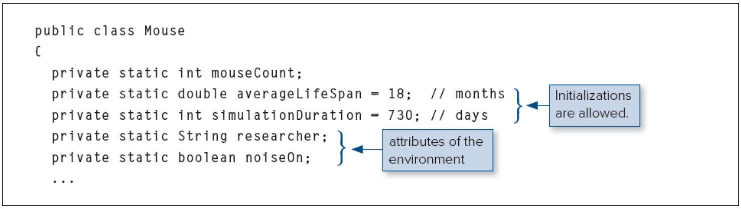
`averageLifeSpan` would keep track of the average life span for all of the mice.

`simulationDuration` would limit the number of simulation iterations.

`researcher` would identify a person in charge of an experiment on the group of mice.

`noiseOn` would indicate the presence or absence of a stressful noise heard by all the mice.

To see how these static variables would be declared within a Mouse program, see Figure 7.21. If you used instance variables for mouseCount, averageLifeSpan, and so on, each individual mouse object would have its own copy of that data. So if there were 100 total mice, each of the 100 mice would store the value 100 in its own mouseCount variable, the average life span value in its own averageLifeSpan variable, and so on. This would mean that every time a new mouse was born or died or aged a year, you would have to update 100 separate copies of mouseCount, averageLifeSpan, and so on—all with exactly the same information. What a waste of effort! Why not just do it once and let everyone write and read the same common data? If mouseCount, averageLifeSpan, and so on are static variables, all mouse objects can write to and read from a single record of each of these pieces of information. An outsider can access these class properties by just prefixing the class name to an appropriate static method. It’s neither necessary nor desirable to go through a particular instance to get to this group information.



```
public class Mouse
{
  private static int mouseCount;
  private static double averageLifeSpan = 18;  // months
  private static int simulationDuration = 730; // days
  private static String researcher;
  private static boolean noiseOn;
  ...
```

Figure 7.21 Static variable declarations in an enhanced `Mouse` class

Default Values

Static variables use the same default values as instance variables:

Static Variable’s Type	Default Value
integer	0
floating point	0.0
`boolean`	`false`
reference	`null`

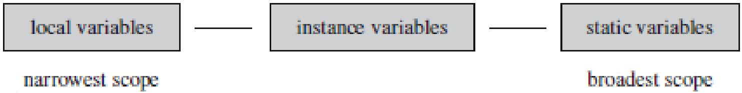
It follows that the default values for Figure 7.21’s static variables are:

```
mouseCount = 0
averageLifeSpan = 0.0
simulationDuration = 0
researcher = null
noiseOn = false
```

Presumably, the program updates mouseCount and averageLifeSpan as it runs. The default values of averageLifeSpan and simulationDuration are zero like mouseCount, but in Figure 7.21, the defaults don’t apply because the declarations include initializations. Even though we expect the program to recompute averageLifeSpan, we initialize it to provide documentation of what we think is a reasonable value (18). We also initialize simulationDuration (to 730) even though we expect the program to reassign simulationDuration with a user-entered value. Presumably, the program prompts the user to enter the number of days to simulate. With appropriate code, the user might be invited to enter −1 to get a “standard” 730-day simulation.

Scope

Let’s now compare static variables, instance variables, and local variables in terms of their scopes. You can access a static variable from anywhere within its class. More specifically, that means you can access static variables from instance methods as well as from static methods. That contrasts with instance variables, which you can access only from instance methods. Thus, static variables have broader scope than instance variables. Local variables, on the other hand, have narrower scope than instance variables. They can be accessed only within one particular method. Here is the scope continuum:


local variables
instance variables
static variables
narrowest scope
broadest scope

Having narrower scope for local variables might seem like a bad thing because it's less "powerful," but it's actually a good thing. Why? Narrower scope equates to more encapsulation, and as you learned in Chapter 6, encapsulation means you are less vulnerable to inappropriate changes. Static variables, with their broad scope and lack of encapsulation, can be accessed and updated from many different places, and that makes programs hard to understand and debug. Having broader scope is necessary at times, but in general, you should try to avoid broader scope. We encourage you to prefer local variables over instance variables and instance variables over static variables.

7.11 Static Methods

Static methods, like static variables, relate to the class as a whole, and they don't relate to individual objects. As such, if you need to perform a task that involves the class as a whole, then you should implement and use a static method. In Chapter 5, you used static methods defined in the Java API Math class; for example, Math.round and Math.sqrt. Now you'll learn how to write your own static methods. Static methods often access static variables, and in writing your own static methods, you'll get an opportunity to see how to access static variables you've defined.

The PennyJar Program

We'd like to present static method syntax details in the context of a complete program. The program we have in mind is a penny jar program . . .

Jordan (author John's daughter) and

Addie (Jordan's BFF) received penny jars for Christmas. They decided to wait until they accumulated a combined total of $40 before breaking open their jars and spending their savings on a huge, ten-pound Hershey bar. They want a program that models their venture. Specifically, they want a program that stores the number of pennies in individual penny jars and also stores the total number of pennies in the collection of all the penny jars. Figure 7.22 shows a *Unified Modeling Language* (UML) class diagram for a -PennyJar class that does just that. The class diagram's middle section contains the program's three variables:

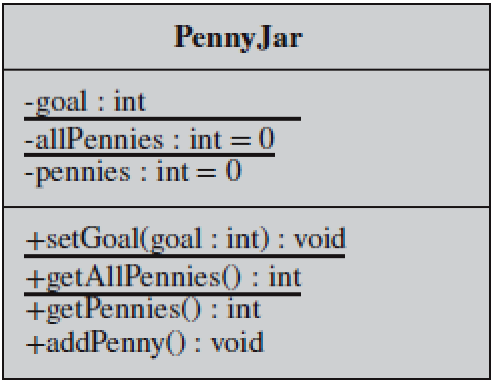

PennyJar
-goal : int
-allPennies : int = 0
-pennies : int = 0
+setGoal(goal : int) : void
+getAllPennies() : int
+getPennies() : int
+addPenny() : void

Figure 7.22 Class describing penny jars

individually and as a group

- The goal variable is the target number of pennies to be saved for all penny jars combined. Because goal is an attribute shared by all the penny jars, it's a static variable. In the UML class diagram, you can tell that goal is a static variable because it's underlined (as you may recall, UML standards suggest that you underline class members).
- The allPennies variable stores the total pennies accumulated so far in all the jars. Because allPennies is an attribute of all the penny jars, it's a static variable and therefore, it is underlined in the UML class diagram. Note that the class diagram shows allPennies being initialized to 0. Although allPennies would have been initialized to 0 by default, the

explicit initialization is a form of self-documentation, making it clear that the programmer needs allPennies to start at 0 for the program to work properly.

- The pennies variable is an instance variable. It keeps track of the number of pennies in one jar.

The class diagram's bottom section contains the program's four methods:

- The setGoal method is a mutator method in that it sets/assigns the value of the goal variable. Because setGoal deals only with classwide data (goal is a static variable), setGoal is a static method. In the UML class diagram, you can tell that setGoal is a static method because it's underlined.
- The getAllPennies method retrieves the value of the allPennies static variable. Because getAllPennies deals only with

class-wide data, getAllPennies is a static method and is also underlined in the UML class diagram.

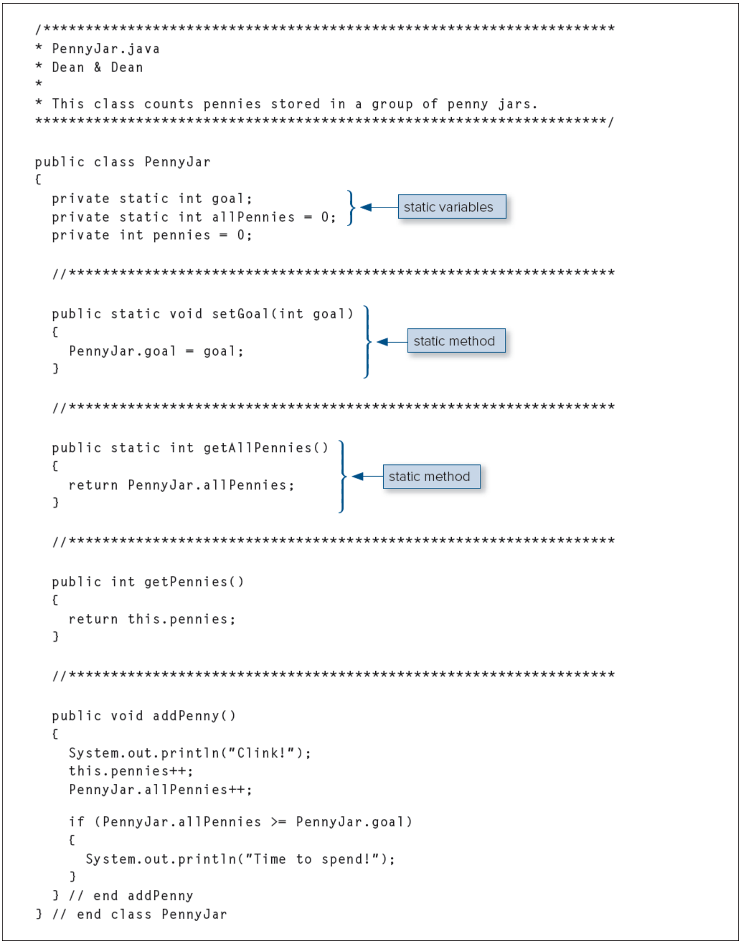
- The getPennies method is an accessor instance method in that it retrieves the value of the pennies instance variable from one particular penny jar.
- The addPenny method is another instance method. It simulates adding a penny to a jar.

How to Define Static Methods

Having introduced the PennyJar program at a high level, it's now time to dig into static method details. To define a static method, use this syntax for the method's heading:

> *private-or-public* static *return-type method-name* (*parameters*)

It's the same syntax that's used for instance methods, except that static appears on the left of the return type. Figure 7.23 shows the source code for the PennyJar program’s PennyJar class. Note that its static methods, setGoal and getAllPennies, have the static modifier in their headings. As with instance methods, most static methods should use the public access modifier (we’ll discuss private methods in the next chapter).



```
/*********************************************************************
* PennyJar.java
* Dean & Dean
*
* This class counts pennies stored in a group of penny jars.
*********************************************************************/

public class PennyJar
{
  private static int goal;
  private static int allPennies = 0;
  private int pennies = 0;

  //******************************************************************

  public static void setGoal(int goal)
  {
    PennyJar.goal = goal;
  }

  //******************************************************************

  public static int getAllPennies()
  {
    return PennyJar.allPennies;
  }

  //******************************************************************

  public int getPennies()
  {
    return this.pennies;
  }

  //******************************************************************

  public void addPenny()
  {
    System.out.println("Clink!");
    this.pennies++;
    PennyJar.allPennies++;

    if (PennyJar.allPennies >= PennyJar.goal)
    {
      System.out.println("Time to spend!");
    }
  } // end addPenny
} // end class PennyJar
```

static variables

static method

static method

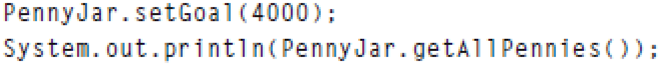
Figure 7.23 `PennyJar` class, which illustrates both instance members and class members

Page 309

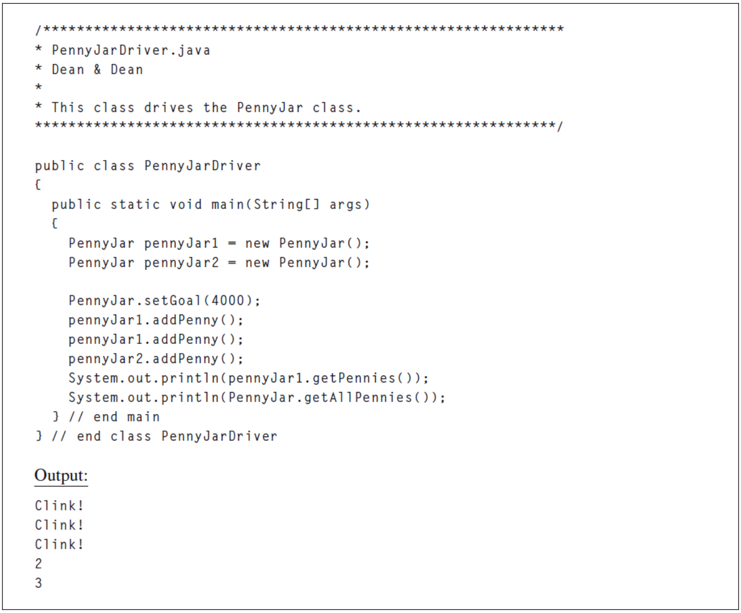
How to Access Class Members

Normally, to access a class member,

you should prefix the class member with the class member's class name and then a dot. Prefixing a class member with its class name and then a dot should sound familiar because you've done that with Math class members like Math.round() and Math.PI. Within Figure 7.23's setGoal and getAllPennies methods, note how the static variables goal and allPennies are accessed with PennyJar dot prefixes—PennyJar.goal and PennyJar.allPennies. For another example, take a look at the PennyJarDriver class in Figure 7.24. In particular, note how its main method calls static methods setGoal and getAllPennies using PennyJar dot prefixes:



```
PennyJar.setGoal(4000);
System.out.println(PennyJar.getAllPennies());
```



```
/*******************************************************************
* PennyJarDriver.java
* Dean & Dean
*
* This class drives the PennyJar class.
*******************************************************************/

public class PennyJarDriver
{
  public static void main(String[] args)
  {
    PennyJar pennyJar1 = new PennyJar();
    PennyJar pennyJar2 = new PennyJar();

    PennyJar.setGoal(4000);
    pennyJar1.addPenny();
    pennyJar1.addPenny();
    pennyJar2.addPenny();
    System.out.println(pennyJar1.getPennies());
    System.out.println(PennyJar.getAllPennies());
  } // end main
} // end class PennyJarDriver
```

Output:

```
Clink!
Clink!
Clink!
2
3
```

Figure 7.24 Driver for the `PennyJar` class in Figure 7.23

These examples show class members being accessed from static methods (setGoal, getAllPennies, and main are all static methods because they all use the static modifier in their headings). Because class members are supposed to be available to the entire class, they can, of course, be accessed from static methods. But be aware that individual objects can also access class members.

Thus, instance methods and constructors (both of which are associated with individual objects) can access class members as well. For example, within the PennyJar program's addPenny instance method, note how the allPennies and goal static variables are accessed:

```
if (PennyJar.allPennies >= PennyJar.goal)
```

Note that it's sometimes legal to omit the class name dot prefix when accessing a class member. In accessing a class member, if the class member is in the same class as the class from which it's accessed, the compiler will allow you to omit the class name dot prefix. For example, the above code fragment accesses allPennies and goal, which are declared in the PennyJar class. Because the code fragment itself comes from the PennyJar class, it would be legal to rewrite the code like this:

```
if (allPennies >= goal)
```

Despite the legality of sometimes being able to omit the class name dot prefix when accessing a class member, we recommend that you always include the class name dot prefix because it's a form of self-documentation. It alerts the person reading the code to the fact that the accessed member is special—it deals with classwide information.

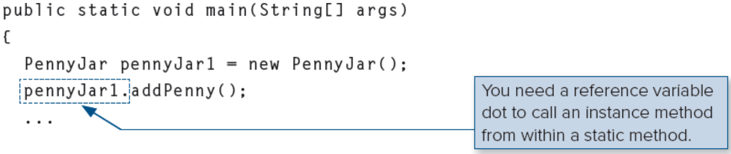
Another reason for always including the class name dot prefix for static variables is that it can be used to avoid ambiguity. If you attempt to access a static variable in a place that has a local variable or parameter declared with the same name as the static variable, then to distinguish the static variable from the local variable or parameter with the same name, you must prefix the static variable with its class name and a dot. If you don't do that, then the compiler will bind your access attempt to the local variable or parameter and not to the static variable. For example, the setGoal method, copied below for your convenience, uses two goal variables—one is the static variable and one is a parameter. To resolve the ambiguity, the static variable must be prefixed with PennyJar dot:

```
public static void setGoal(int goal)
{
  PennyJar.goal = goal;
} // end setGoal
```

This identifies a static variable and distinguishes it from a parameter having the same name.

Calling an Instance Method from Within a Static Method

If you're within a static method, you'll get a compilation error if you attempt to directly access an instance member. To access an instance member, you first must have an object, and then you access the object's instance member by prefixing it with the object's reference variable. The reference variable is often referred to as the *calling object.* Does all that sound familiar? The main method is a static method, and you've been calling instance methods from main for quite a while now. But whenever you did that, you first instantiated an object and assigned the object's reference to a reference variable. Then you called the instance method by prefixing it with the reference variable and a dot. Figure 7.24's main method shows what we're talking about:



```
public static void main(String[] args)
{
  PennyJar pennyJar1 = new PennyJar();
  pennyJar1.addPenny();
  ...
```

You need a reference variable dot to call an instance method from within a static method.

If you attempt to access an instance method directly from within a static method without using a reference variable dot prefix, you'll see an error message like this:

```
Non-static <method name> cannot be referenced from a static context
```

That error message is very common (you've probably seen it many times) because it's easy to forget to prefix instance method calls with a reference variable. When veteran programmers see it, they know what to do; they make sure to prefix the instance method call with a calling object's reference variable. But when beginning programmers see the error message, they often compound the error by trying to "fix" the bug inappropriately. More specifically, when confronted with the non-static method error message, a beginning programmer often will change the offending instance method to a static method, by inserting static in the method's heading. (In the PennyJar program, addPenny would be changed to a static method). They then get the non-static member error message for any instance variables within the method. They then compound the problem further by changing the method's instance variables to static variables. (In the PennyJar program, addPenny's pennies variable would be changed to a static variable). With that change in place, the program compiles successfully and the beginning programmer is happy as a lark, ready to slay the next dragon. Unfortunately, that type of solution leads to a worse problem than a compilation error. It leads to a logic error.

As you know, if a class's member relates to one object rather than to the class as a whole, you should make it an instance member. If you do as described above and "fix" a bug by changing instance members to class members, you can get your program to compile and run. And if you have only one object, your program might even produce a valid result. But if you have more than one object, either now or in the future, then with static variables, the objects will share the same data. If you change one object's data, you'll simultaneously change all the objects' data and that would normally be deemed incorrect. For the PennyJar program, all the PennyJar objects would share the same pennies value, so it would be impossible to keep track of the money in individual penny jars.

When to Use Static Methods

When should you make a method a static method? The general answer is, "when you need to perform a task that involves the class as a whole." But let's get more specific. Here are situations where using static methods is appropriate:

1. If you have a method that uses static variables, calls static methods, or both, then it's a good candidate for being a static method. For example, Figure 7.23's getAllPennies is a static method because it retrieves the value allPennies, a static variable. Warning: If in addition to accessing class members, the

method accesses instance members, then the method must be an instance method, not a static method.

2. The main method is the starting point for all programs and, as such, it gets executed prior to the instantiation of any objects. To accommodate that functionality, you're required to make the main method a static method. If your main method is rather long and contains well-defined subtasks, you should consider trying to implement those subtasks with their own methods. For an example of that, see the RollDice program in the Interlude "mini-chapter" before Chapter 6. You should use static for the subtask methods, thereby making them static methods. By making them static methods, it's easy for main to call them (just prefix them with class name dot, rather than having to instantiate an object first and then prefix them with reference variable dot).

3. If you have a general-purpose method that stands on its own, make it a static method. By standing on its own, we mean that the method is not related to a particular object. Such methods are called *utility methods*. You’ve seen examples of utility methods, like Math.round and Math.sqrt, in the Math class. In the next chapter, you’ll learn how to write your own utility methods within the context of a general-purpose utility class.

7.12 Named Constants

Using names instead of hard-coded

values makes a program more self-documenting. When a constant value is needed in more than one place in the block of code, establishing the value at one place at the beginning of that block minimizes the chance of inconsistency. In Java, you can define named constants at several levels of scale.

Local Named Constants—A Review from Chapter 3

At the most microscopic level, you can define local named constants. Back in Figure 3.5 of Chapter 3, we defined two local named constants, FREEZING_POINT, and CONVERSION_FACTOR, to self-document the Fahrenheit-to-Celsius conversion formula in a simple program that did nothing more than make a temperature conversion. Usually, we embed this kind of activity in some larger program by putting it in a helper method like this:

```
private double fahrenheitToCelsius(double fahrenheit)
{
  final double FREEZING_POINT = 32.0;
  final double CONVERSION_FACTOR = 5.0 / 9.0;

  return CONVERSION_FACTOR * (fahrenheit - FREEZING_POINT);
} // end fahrenheitToCelsius
```

The local named constants in this method make the code easier to understand.

Instance Constants—A Review from Section 7.8

At the next higher level of scale, sometimes you want a constant that's a permanent property of an object and accessible to all instance methods associated with that object. Those constants are called *instance named constants*, or, more simply, *instance constants*. Here's an example instance constant declaration that identifies a permanent property of a Person object:

```
public final String SOCIAL_SECURITY_NUMBER;
```

An instance constant declaration differs from a local named constant declaration in three ways: (1) An instance constant declaration should appear at the top of the class definition, rather than within a method; (2) an instance constant declaration is preceded by a public or private access modifier; and (3) although it's legal to initialize an instance constant in a declaration, it's more common to initialize it in a constructor.

Static Constants

At the next higher level of scale, sometimes you want a constant that's the same for all objects in a class. In other words, you want something that's like a static variable, but it's constant. Those constants are called static named constants, or, more simply, *static constants.*[4] In Chapter 5, you learned about two static constants defined in the Java API Math class, PI and E. Now you'll learn how to write your own static constants. To declare a static constant, use this syntax:

> *private-or-public* static final *type variable-name* = *initial-value*;

A static constant declaration differs from an instance constant declaration in two ways: (1) A static constant includes the static modifier; and (2) A static constant should be initialized as part of its declaration.5 If you attempt to assign a value to a static constant later on, that generates a compilation error.

As with an instance constant, a static constant declaration should be preceded by a public or private access modifier. If the constant is needed only within the class (and not outside the class), you should make it private. This allows you to modify the constant without upsetting somebody else who previously elected to use your constant in one of their programs. However, if you want the constant to be available to other classes, it's appropriate to make it public. It's safe to do that because the final modifier makes it immutable (unchangeable).

The following Human class contains a NORMAL_TEMP named constant. We make it a static constant (with the static and final modifiers) because all Human objects have the same normal temperature of 98.6° Fahrenheit. We make it a private static constant because it is needed only within the Human class.

```
public class Human
{
  private static final double NORMAL_TEMP = 98.6;
  private double currentTemp;
  ...
  public boolean isHealthy()
  {
    return Math.abs(currentTemp - NORMAL_TEMP) < 1;
  } // end isHealthy
  public void diagnose()
  {
    if ((currentTemp - NORMAL_TEMP) > 5)
    {
      System.out.println("Go to the emergency room now!");
      ...
} // end class Human
```

Let’s summarize when you should use the three different types of named constants. Use a local named constant if the constant is needed within only one method. Use an instance constant if the constant describes a permanent property of an object. And use a static constant if the constant is a permanent property of the collection of all the objects in the class or of the class in general.

Positions of Declarations

Now for some coding-style issues. We recommend putting all static constant declarations above all instance constant declarations. Putting declarations at the top makes them stand out more, and it’s appropriate for static constants to stand out the most because they have the broadest scope. Likewise, we recommend putting all static variable declarations above all instance variable declarations. *Field* is the generic term for a static constant, instance constant, static variable, or instance variable. We recommend that you put all your field declarations at the top, above all your constructor and method declarations. Here is the preferred sequence of declarations within a given class:

static constants
instance constants
static variables
instance variables
constructors
mutator and accessor methods (doesn't matter which type is first)
other methods (doesn't matter which type is first—static methods or instance methods)

7.13 Problem Solving with Multiple Driven

Classes

We started this book simply, and we are gradually adding complexity as we go. In Chapters 1 through 5, we showed you programs that contain only one class and one method (the main method). In Chapters 6 and 7, we've been showing you programs that contain two classes: (1) a driver class, which contains a single main method; and (2) a driven class, which typically contains several methods.

Break up a big problem into isolated simpler problems.

So far, we've used only one driven class to keep things simple, but in the real world, you'll often need more than one driven class. That's because most real-world systems are heterogeneous—they contain mixtures of different types of things. For each different type of thing, it's appropriate to have a different class. Having more than one driven class allows you to partition a complicated problem into several simpler problems. That lets you focus on one type of thing at a time. When you finish working on one type of thing, you can move onto another type of thing. In this step-by-step fashion, you can gradually build up a large program.

It’s no big deal to drive more than one driven class from a single driver. In fact, you saw us do it back in Chapter 5, when statements in a single main method called methods from more than one wrapper class, like Integer and Double. The only thing to remember is when compiling the driver, the compiler must be able to find all the driven classes. If they are prebuilt classes, they must be part of the java.lang package or you must import them. If they are classes that you’ve written yourself, they should be in the same directory as your driver.[6]

Summary

- When you declare a reference variable, the JVM allocates space in memory for holding a reference to an object. At that point, there is no memory allocation for the object itself.
- Assigning one reference variable to another does not clone an object. It just makes both reference variables refer to the same object and gives that object an alternate name—an alias.
- To create a separate object, you must use Java's new operator. To make a second object be like a first object, copy the first object's instance variable values into the corresponding instance variables

in the second object.

- A method can return an assortment of data originating in a method by returning a reference to an internally instantiated object that contains that data.
- Java's garbage collection program searches for inaccessible objects and recycles the space they occupy by asking the operating system to designate their space in memory as free space.
- If you compare two object references with = =, the result is true if and only if the references point to the same object.
- To see whether two different objects contain similar data, you must write an equals method that individually compares respective instance variable values.
- To swap two variables' values,

you need to store one of the variable’s values in a temporary variable.

- If you pass a reference as an argument, and if the reference parameter’s instance variables are updated, then the update simultaneously updates the reference argument’s instance variables in the calling module.
- If a method returns a reference to an object, you can use what’s returned to call another method in the same statement. That’s method-call chaining.
- To make a program more understandable, you can *overload* a method name by using the same name again in a different method definition that has a different sequence of parameter types. The combination of method name, number of parameters, and

parameter types is called a method’s *signature*.

- A constructor enables you to initialize instance variables separately for each object. A constructor’s name is the same as its class name, and there is no return value specification.
- For a constructor call to work there must be a matching constructor definition, that is, a definition with the same signature.
- If you define a constructor, the default zero-parameter constructor vanishes.
- Use a constructor to initialize instance constants, which represent permanent attributes of individual objects.
- To get help constructing the current object, call an overloaded constructor from within a

constructor, by making the first statement in the constructor be: this(*constructor-argument(s)*).

- Static variables have a static modifier. Use static variables for attributes of the collection of all objects in a class. Use instance variables for the attributes of individual objects.

- Remember that static variables have broader scope than instance variables, and instance variables have broader scope than local variables. To improve encapsulation, you should try to use variables with a narrower scope rather than a broader scope.

- A static method can access class members directly, but it cannot access instance members directly. To access an instance member from a static method, you need to use an object reference variable dot prefix.

- An instance method can directly access class members as well as instance members.

- Use static constants for permanent data that is not associated with any particular object. Static constants use the final and static modifiers.

- Instance constants have a final modifier only. Use instance constants for permanent attributes of individual objects.

Review Questions

§7.2 Object Creation—A Detailed Analysis

1. The statement

```
Car car;
```

allocates space in memory for an object. (T / F)

2. What does the new operator do?

§7.3 Assigning a Reference

3. Assigning one reference variable to another copies the right-side object’s instance variables into the left-side object’s instance variables. (T / F)

4. What is a memory leak?

§7.4 Testing Objects for Equality

5. Consider this code fragment:

```
boolean same;
Car carX = new Car();
Car carY = carX;
same = (carX == carY);
```

6. What is the return type of an equals method?

7. By convention, we use the name equals for methods that perform a certain kind of evaluation. What is the difference between the evaluation performed by an equals method and the == operator?

§7.5 Passing References as Arguments

8. When you pass a reference to a method, you enable the method to modify the referenced object. (T / F)

§7.6 Method-Call Chaining

9. What two things must be included in a method definition so that it may be called as part of a method-call-chaining statement?

§7.7 Overloaded Methods

10. What is it called when you have two or more methods with the same name in the same class?

11. If you want the current object

to call a different method in the same class as the current class, the method call is easy—just call the method directly, with no reference variable dot prefix. (T / F)

§7.8 Constructors

12. What is the return type of a constructor?

13. The name of a constructor must be exactly the same as the name of its class. (T / F)

14. Standard coding conventions suggest that you put constructor definitions after the definitions of all methods. (T / F)

§7.9 Overloaded Constructors

15. If a class's source code contains a single one-parameter constructor, the constructor is overloaded because this one-parameter constructor has the same name as the default zero-parameter constructor. (T / F)

16. Suppose you have a class with two constructors. What are the rules for calling one constructor from the other constructor?

§7.10 Static Variables

Page 317

17. Normally, you should use the private access modifier for static variables. (T / F)

18. When should you declare a variable to be a static variable, as opposed to an instance variable?

19. What are the default values for static variables?

§7.11 Static Methods

20. What would be wrong with adding the following statement to Figure 7.24's main method?

```
PennyJar.addPenny();
```

21. Member access:

a) It is OK to use this in a static method. (T / F)

b) It is OK to use the class name as a prefix when calling a static method. (T / F)

c) Within a main method, it is OK to omit the class name prefix before the name of another static method being called. (T / F)

22. It is legal to access a class member from an instance method and also from a constructor. (T / F)

23. It is legal to directly access an instance member from a static method. (T / F)

§7.12 Named Constants

24. What keyword converts a variable into a constant?

25. If you want a named constant

used by instance methods to have the same value, regardless of which object accesses it, the declaration should include the static modifier. (T / F)

26. A static constant should be initialized within a constructor. (T / F)

27. Suppose you have a grading program that instantiates multiple exam objects from an Exam class. Provide a declaration for a constant minimum passing score. Assume the minimum passing score for all exams is 59.5.

Exercises

---

1. [after §7.2] Given a Car class with these two instance variables:

```
int year;
String make;
```

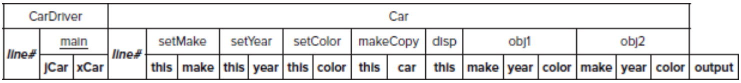
Describe all the operations that

occur when this statement executes:

Car shyanCar = new Car();

2. [after §7.3] Explain garbage collection.

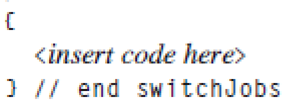
3. [after §7.3] Trace the Car program shown in Figures 7.2 and 7.3 using the following trace setup. The abbreviations in this suggested header keep the trace's width as small as possible.



CarDriver			Car																
line#	main		*line#*	setMake		setYear		setColor		makeCopy		disp	obj1			obj2			
	jCar	xCar		this	make	this	year	this	color	this	car	this	make	year	color	make	year	color	output

4. [after §7.5] Suppose you have an updateAccount method with two parameters, an int variable named yearsAccumulatingInterest and a reference variable named bankAccount. The bankAccount variable's object contains an instance variable named balance. Within the method's body, the yearsAccumulatingInterest parameter gets updated and the bankAccount object's balance gets updated. In a method that calls the updateAccount method, after returning from the updateAccount method, the yearsAccumulatingInterest argument's value is unchanged and the bankAccount argument's balance value is changed. Explain why there is a difference.

5. [after §7.5] Suppose you enhance the Person class in Figure 7.7 by adding another instance variable, jobDescription, and another method, switchJobs. Complete the following additional switchJobs method that swaps the calling object's jobDescription with the passed-in parameter object's jobDescription.



```
public void switchJobs(Person otherPerson)
{
  <insert code here>
} // end switchJobs
```

6. [after §7.6] In JavaFX CSS

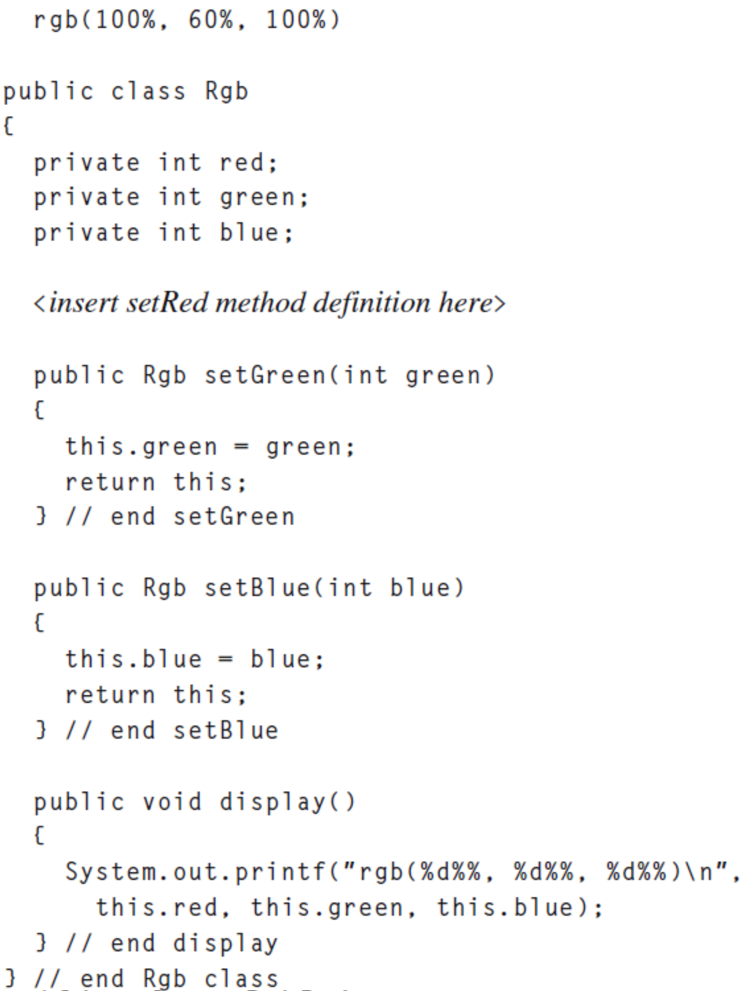
(introduced in Chapter 17), you can form a color by specifying certain amounts of red, green, and blue, with each color’s amount ranging from 0% to 100%. For example, to specify purple, you could use this code:

```
rgb(100%, 60%, 100%)
```

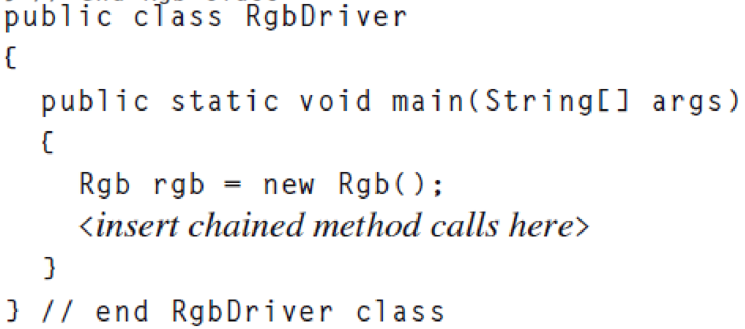
The rgb stands for red, green, and blue. With 100% for red and blue, the resulting color is purple. In the following Rgb and RgbDriver class skeletons, replace the <*insert…*> lines with appropriate code such that the program operates properly. More specifically:

a) In the Rgb class, provide a method definition for the setRed method such that setRed can be called as part of a method-call chain.

b) In the RgbDriver class, provide a single statement that chains calls to the setRed, setGreen, setBlue, and display methods. For your method-call arguments, pass 100 to setRed, 60 to setGreen, and 100 to setBlue. With those argument values, your method-call-chaining statement should print this:

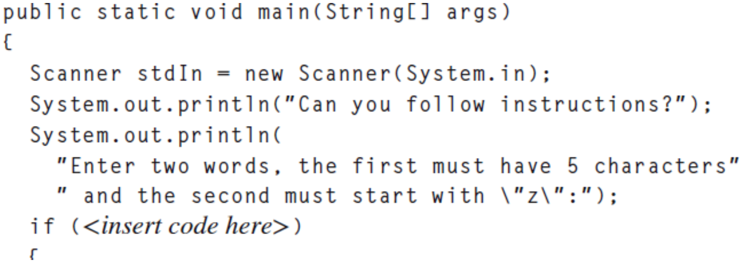


```
  rgb(100%, 60%, 100%)

public class Rgb
{
  private int red;
  private int green;
  private int blue;

  <insert setRed method definition here>

  public Rgb setGreen(int green)
  {
    this.green = green;
    return this;
  } // end setGreen

  public Rgb setBlue(int blue)
  {
    this.blue = blue;
    return this;
  } // end setBlue

  public void display()
  {
    System.out.printf("rgb(%d%%, %d%%, %d%%)\n",
      this.red, this.green, this.blue);
  } // end display
} // end Rgb class
```



```
public class RgbDriver
{
  public static void main(String[] args)
  {
    Rgb rgb = new Rgb();
    <insert chained method calls here>
  }
} // end RgbDriver class
```

7. [after §7.6] Given the following main method, replace the <insert code here> item with code that reads two words and checks for the first one having 5 characters and the second one starting with z. You must use method-call chaining.



```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  System.out.println("Can you follow instructions?");
  System.out.println(
    "Enter two words, the first must have 5 characters"
    " and the second must start with \"z\":");
  if (<insert code here>)
  {
```

8. [after §7.7] Overloaded Methods

a) In this chapter's Height class modify the two-parameter setHeight method to see if the units value is one of the following: "in", "ft", "mm", "cm", or "m." If it is, update the units field and return the boolean value, true. Otherwise print

"Unrecognizable units: <entered units>" and return false.

b) Modify the corresponding HeightDriver class so that it calls the Height's print method only if a previous call to the two-parameter setHeight method returns true.

9. [after §7.8] Provide a standard three-parameter constructor for a class named CellPhone. The class contains three instance variables—manufacturer, model, and price. The constructor simply assigns its three parameters to the three instance variables.

10. [after §7.9] Overloaded Constructors

a) Implement a Weight class with a two-parameter setWeight method that initializes weight and units variables and a one-parameter setWeight method that uses a default “kg” for units. Provide corresponding two-parameter and one-parameter constructors and a print method, which displays weight and units values. Minimize the total number of statements by having the one-parameter constructor call the one-parameter setWeight method and the two-parameter constructor call the two-parameter setWeight method.

b) Provide a main method for a WeightDriver class such that the new main method uses one of the new constructors from part a) to generate this output:

```
120 lbs
```

11. [after §7.9] Assume that a BodyWeight class contains only one setWeight method—a two-parameter version with the parameters double weight and String units. Write two constructors for the BodyWeight class, one with one argument (double weight), and the other with two arguments (double weight and String units). For the one-argument constructor, use the default of “kg” for units. Avoid code redundancy. That is, have the one-parameter constructor call the two-parameter constructor, and have the two-parameter constructor call the two-parameter setWeight method.

12. [after §7.9] Assume that the following two classes are compiled and run. What is the output?

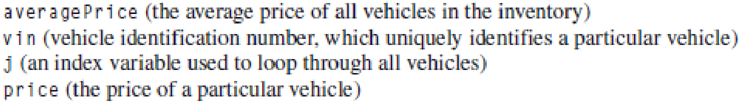
You don’t have to provide a formal trace—you just have to show the output. But to determine the output, you'll need to do an informal trace either in your head or with a few notes. You are encouraged to verify your answer by running the program on a computer. Running the program is for verification purposes, not for coming up with the answer in the first place. You should be able to understand how the program works by just examining its code.

```
public class ConfuseMeDriver
{
  public static void main(String[] args)
  {
    ConfuseMe confuse = new ConfuseMe();
    confuse.display();
  }
} // end ConfuseMeDriver class

public class ConfuseMe
{
  private int x = 40;

  public ConfuseMe()
  {
    this(25);
    System.out.println(this.x);
  }

  public ConfuseMe(int x)
  {
    System.out.println(this.x);
    System.out.println(x);
    this.x = 10;
    x = 50;
  }

  public void display()
  {
    int x = 30;
    display(x);
    System.out.println(x);
  }

  public void display(int x)
  {
    x += 15;
    System.out.println(x);
  }
} // end ConfuseMe class
```

13. [after §7.10] An object's

methods cannot alter the value of one of its class's variables that has private static modifiers. (T / F)

14. [after §7.10] Given a program that keeps track of automobiles in an automobile dealer's inventory, for each of the following program variables, specify whether it should be a local variable, an instance variable, or a static variable.



`averagePrice` (the average price of all vehicles in the inventory)
`vin` (vehicle identification number, which uniquely identifies a particular vehicle)
`j` (an index variable used to loop through all vehicles)
`price` (the price of a particular vehicle)

15. [after §7.10] In general, why should you prefer instance variables over static variables, and why should you prefer local variables over instance variables?

16. [after §7.11] If a method accesses a static variable, the method:

a) must be a local method

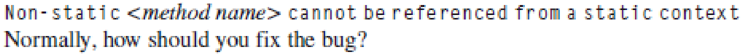
b) must be an instance

method

c) must be a static method

d) can be either a static method or an instance method—it depends on other factors

17. [after §7.11] If you attempt to directly access an instance method from within a static method, you'll see an error message like this:

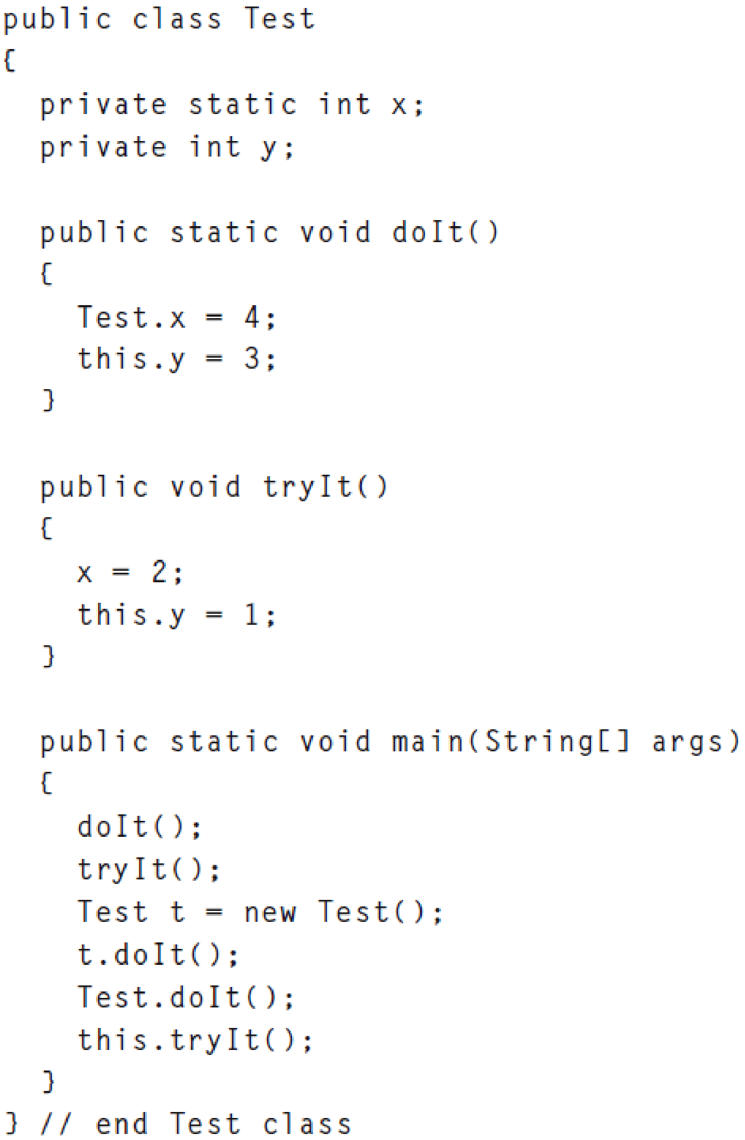


```
Non-static <method name> cannot be referenced from a static context
```

Normally, how should you fix the bug?

Normally, how should you fix the bug?

18. [after §7.11] Consider the following program:



```
public class Test
{
  private static int x;
  private int y;

  public static void doIt()
  {
    Test.x = 4;
    this.y = 3;
  }

  public void tryIt()
  {
    x = 2;
    this.y = 1;
  }

  public static void main(String[] args)
  {
    doIt();
    tryIt();
    Test t = new Test();
    t.doIt();
    Test.doIt();
    this.tryIt();
  }
} // end Test class
```

Mark all of the lines of code that have a compilation error.

For each compilation-error line, explain why it is incorrect.

For each compilation error, just provide the reason that the error occurs. Do not try to solve the problem by attempting to show a “fixed” version of the program.

19. [after §7.12] Write appropriate declarations for the following named constants. In each case, decide whether to include the keyword, static, and whether to include initialization in the declaration. Also, make each constant as easily accessible as possible, consistent with protection from inadvertent corruption.

a) The format string, "%-16s %,9.3f%,12.0f%(,12.3f\n", to use in several printf statements in a single method.

b) The proton electron mass ratio: 1836.15267389

c) The year of birth of a person.

20. [after §7.12] Why is it safe to declare named constants public?

Review Question Solutions

1. False. It just allocates memory for a reference variable.

2. The new operator allocates memory for an object and returns the address of where that object is stored in memory.

3. False. Assigning a reference variable to another reference variable causes the address in the right side's reference variable to be put into the left side's reference variable. And that makes both reference variables refer to the same object.

4. A memory leak is when an

inaccessible object is allowed to persist and use up space in a computer’s memory.

5. The final value of same is true.

6. The return type of an equals method is boolean.

7. The == operator compares the values of two variables of the same type. If the variables are reference variables, == compares their addresses to see if they refer to the same object. A typical equals method compares the values of all the instance variables in the object referred to by its parameter with the values of corresponding instance variables in the object that called it. It returns true only if all corresponding instance variables have the same values.

8. True. The reference gives the method access to the reference’s object.

9. For a method to be called as part of a method-call-chaining statement, include these things:

- Within the method body, specify return reference-variable;
- Within the method heading, specify the reference variable's associated class as the return type.

10. If you have two or more methods with the same name in the same class, they're called overloaded methods.

11. True.

12. A constructor does not have a return type and it does not use a return statement, but when you call a constructor, new returns a reference to the constructed object.

13. True.

14. False. Standard coding

conventions suggest that you put constructor definitions before all other method definitions.

15. False. There is only one constructor, because if a class contains a programmer-defined constructor, then the compiler does not provide a default constructor.

Page 323

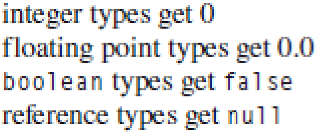
16. To call the other constructor, you must insert this statement as the very first statement in the calling constructor's body:

> this(*arguments-for-target-constructor*);

17. True.

18. You should declare a variable to be a static variable, as opposed to an instance variable, if the variable holds data that is associated with the class as a whole. You should use static variables to describe properties of a class's objects that need to be shared by all of the objects.

19. The default values for static variables are the same as they are for instance variables of the same type. Here are the default values:



integer types get 0
floating point types get 0.0
`boolean` types get `false`
reference types get `null`

20. Because there's no static modifier, addPenny is an instance method. The PennyJar.addPenny(); call uses a PennyJar dot prefix. It's illegal to use a class name (PennyJar) as a prefix for an instance method call. To call an instance method, you need to use a reference variable dot prefix.

21. Member access:

a) False. You cannot use this in a static method.

b) True. You can always use

the class name as a static method prefix.

c) True, if the main method is “merged” into the same class as the other method.

False, if the other method is in a different class.

Including the class name prefix allows you to move the main method to another class later.

22. True. You can access a class member from an instance method and also from a constructor—just prefix the class member with the class name.

23. False. You can access an instance member from a static method only if you prefix the method name with a reference to a particular object.

24. The keyword final converts a variable into a constant.

25. True. Use static to make a constant be the same for all objects.

26. False. A static constant should normally be initialized as part of its declaration. If it is assigned a value later on, including within a constructor, it generates a compilation error.

27. Minimum passing score declaration:

```
private static final double MIN_PASSING_SCORE = 59.5;
```

---

[1] The concept of garbage collection appeared earlier in the LISP programming language. LISP is a compact language that in principle can do anything, but its primary application is artificial intelligence.

[2] You probably already know what a “rat’s nest” is—a tangled mess. But how about “Cerberean”? In Greek mythology, Cerberus is a vicious,

three-headed dog creature that guards the entrance to Hades (the world of the dead). We say our return statement might appear to be a Cerberean rat's nest because it's complicated and it has three parts. Which would you rather meet in a dark alley—a vicious, three-headed dog creature or a complicated return statement?

[3] If you try to define a constructor with a return type specification, the compiler will not recognize it as a constructor and will think it is an ordinary method instead.

[4] In their API library, Oracle uses the term "constant field" rather than "static constant." We prefer the term "static constant" because it's more descriptive, and we need to distinguish between static constants and instance constants.

[5] Although relatively rare, it's legal to declare a static constant as part of a

static initialization block. For details on *initialization blocks* see https://docs.oracle.com/javase/tutorial/java/javaOO/initial.html.

[6] It's possible to put your own classes in your own packages in separate directories and import them as you import prebuilt classes. You can learn how to do this in Appendix 4 of this book. However, if all your driven classes are in the same directory as your driver class, it's not necessary to package and import them, and we assume this to be the case for the code that we develop and present in this text.

Software Engineering

Objectives

- Develop good coding style.
- Understand preconditions and postconditions.
- Learn how to generate documentation like Java’s API documentation.

- Learn how to simplify complicated algorithms by encapsulating subordinate tasks.
- Learn how to adapt a user’s point of view.
- Distinguish use of instance variables and local variables.
- Analyze problems by separating concerns.
- Learn when and how to use a top-down design strategy.
- Learn when and how to use a bottom-up design strategy.
- Resolve to use prewritten software whenever feasible.
- Recognize role of prototyping.
- Develop habit of frequent and thorough testing.
- Avoid unnecessary use of the this prefix.

- Recognize appropriate uses of static methods.

Outline

8.1 Introduction 325

8.2 Coding-Style Conventions 325

8.3 Documentation for Outsiders 334

8.4 Helper Methods 338

8.5 Encapsulation (with Instance Variables and Local Variables) 342

8.6 Recognizing the User’s Point of View 344

8.7 Design Philosophy 345

8.8 Top-Down Design 350

8.9 Bottom-Up Design 359

8.10 Case-Based Design 361

8.11 Iterative Enhancement 361

8.12 Merging the Driver Method into the Driven Class 363

8.13 Accessing Instance Variables Without Using this 365

8.14 Writing Your Own Utility Class 366

8.15 Problem Solving with the API Calendar Class (Optional) 368

8.16 GUI Track: Problem Solving with CRC Cards (Optional) 370

8.1 Introduction

In Chapters 6 and 7, we looked mostly

at the “science” of Java programming—how to declare objects, define classes, define methods, and so on. In this chapter, we’ll be looking more at the “art” and “practice” of Java programming—how to design and develop a program, and how to make it easy to read. The practice of programming is nicely summed up in the term *software engineering,* where software engineering is:1

> **1.** The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
>
> **2.** The study of approaches as in 1.

We start the chapter with an in-depth discussion of coding-style conventions that make code more readable to people who need to know how it works. Then we backtrack a bit and describe techniques that make a block of code understandable to people who do not need to know how it works, only what it does. Next, we show how to divide a large task into a set of smaller tasks by delegating some of the work in a method to other methods. We discuss encapsulation, one of the cornerstones of proper object-oriented programming (OOP) design.

Next, we describe alternative design strategies—top-down, bottom-up, and case-based. As you work with something, your understanding of it improves, and we suggest that you plan to redesign continuously with more sophistication in an evolutionary process called *iterative enhancement*. We emphasize that you'll be happier and your product will be better if you test thoroughly and frequently as you go along.

To facilitate modular testing, we show how you can include a main method in each class. Up to now, we've made heavy use of this to emphasize that each execution of an instance method is uniquely tied to a particular object, but you can streamline your code by omitting this when there is no ambiguity. Sometimes it's useful to create your own utility class with static methods that methods in other classes can call easily. An optional section shows you how to use Java's Calendar class for difficult time and date problems. Another optional section shows you how to use simple graphics to construct a handy organizational tool called *CRC cards*.

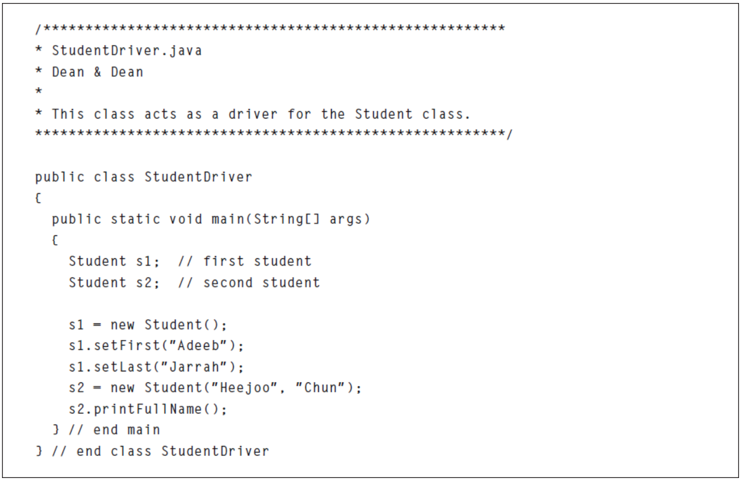
8.2 Coding-Style Conventions

We'll now present some guidelines for

coding style. We've mentioned and illustrated many of these style guidelines previously, so much of this section should be review. We'll provide more guidelines later, as we describe more Java. For a complete list of all the coding-style guidelines used in this book, refer to Appendix 5, "Java Coding-Style Conventions." The coding-style conventions we use are for the most part a simplified subset of the style conventions archived on Oracle's Java Code Conventions website.[2] If you have a style question that is not addressed in Appendix 5, refer to Oracle's website.

We realize there are some style issues where there is legitimate disagreement over the best way to do things. Many different standards exist. Oracle attempts to choose the best conventions from among the commonly used conventions. We attempt to do the same. If you're reading this book as part of a course and your teacher disagrees with the book's style conventions or Oracle's style conventions, please follow your teacher's guidelines.

We'll illustrate coding-style conventions by referring to the Student program in Figure 8.1 and Figures 8.2a and 8.2b. This program is a modified version of the Student program at the back of the "Java Coding-Style - Conventions" appendix.



```
/*********************************************************
* StudentDriver.java
* Dean & Dean
*
* This class acts as a driver for the Student class.
*********************************************************/

public class StudentDriver
{
  public static void main(String[] args)
  {
    Student s1;  // first student
    Student s2;  // second student

    s1 = new Student();
    s1.setFirst("Adeeb");
    s1.setLast("Jarrah");
    s2 = new Student("Heejoo", "Chun");
    s2.printFullName();
  } // end main
} // end class StudentDriver
```

Figure 8.1 StudentDriver class

Prologue Section

Note the boxed text at the tops of Figures 8.1 and 8.2a. They're called *prologues.* Include a prologue section at the top of each file. The prologue contains these things, in this order:

- line of asterisks
- filename
- programmer name(s)
- blank line with one asterisk
- description
- line of asterisks
- blank line

Enclose the prologue in a /*. . .*/ comment, and to make the prologue look like a box, insert an asterisk and a space in front of the filename, programmer name, blank line, and description lines.

Field Declarations and Initializations

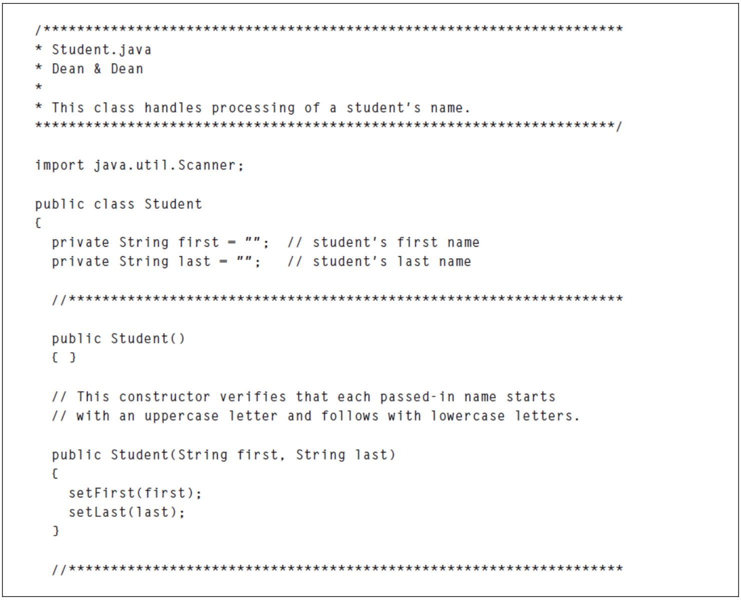
For each class, declare and/or initialize your class's fields at the top of the class's body (a field is a generic term for a static constant, an instance constant, a static variable, or an instance variable). Below your field declarations/initializations, you should provide a blank line, a line of asterisks, and another blank line.

Method Descriptions

Note the descriptions above one of the constructors in Figure 8.2a and the methods in Figure 8.2b. Put things in this order above each method:

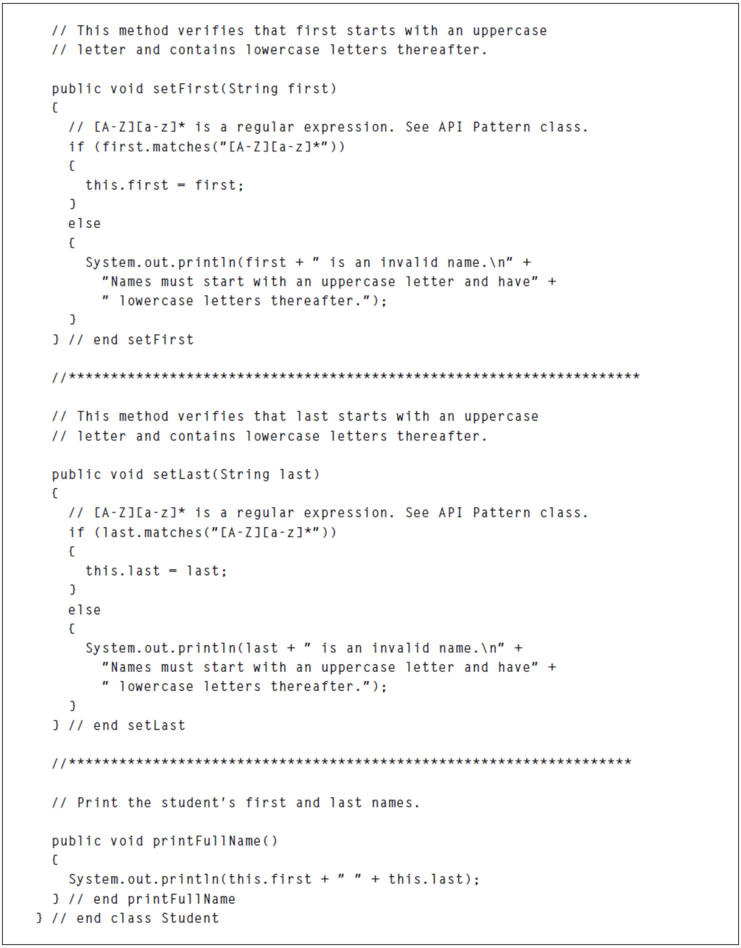
- blank line
- line of asterisks
- blank line
- description
- blank line

For short obvious methods, it's OK to omit the method description. Between short constructors and between short accessor and mutator methods, it’s also OK to omit the line of asterisks.



```
/*********************************************************************
* Student.java
* Dean & Dean
*
* This class handles processing of a student's name.
*********************************************************************/

import java.util.Scanner;

public class Student
{
  private String first = "";  // student's first name
  private String last = "";   // student's last name

  //*******************************************************************

  public Student()
  { }

  // This constructor verifies that each passed-in name starts
  // with an uppercase letter and follows with lowercase letters.

  public Student(String first, String last)
  {
    setFirst(first);
    setLast(last);
  }

  //*******************************************************************
```

Figure 8.2a Student class—part A



```
  // This method verifies that first starts with an uppercase
  // letter and contains lowercase letters thereafter.

  public void setFirst(String first)
  {
    // [A-Z][a-z]* is a regular expression. See API Pattern class.
    if (first.matches("[A-Z][a-z]*"))
    {
      this.first = first;
    }
    else
    {
      System.out.println(first + " is an invalid name.\n" +
        "Names must start with an uppercase letter and have" +
        " lowercase letters thereafter.");
    }
  } // end setFirst

  //*******************************************************************

  // This method verifies that last starts with an uppercase
  // letter and contains lowercase letters thereafter.

  public void setLast(String last)
  {
    // [A-Z][a-z]* is a regular expression. See API Pattern class.
    if (last.matches("[A-Z][a-z]*"))
    {
      this.last = last;
    }
    else
    {
      System.out.println(last + " is an invalid name.\n" +
        "Names must start with an uppercase letter and have" +
        " lowercase letters thereafter.");
    }
  } // end setLast

  //*******************************************************************

  // Print the student's first and last names.

  public void printFullName()
  {
    System.out.println(this.first + " " + this.last);
  } // end printFullName
} // end class Student
```

Figure 8.2b Student class—part B

Blank Lines

In general, use blank lines to separate logical chunks of code. In Figure 8.1's StudentDriver class, note the blank lines:

- between the prologue section and the class definition
- right after a method's local variable declarations

It's not shown in the Student program, but for long methods, it's appropriate to insert blank lines between logically separate chunks of code within the method. Also, when a comment line appears within the body of the code, it's nice to have white space above that comment to make it more visible.

Meaningful Names

Use meaningful names for your classes and variables. For example, Student is a good name for the class in Figures 8.2a and 8.2b because the class models a student. Similarly, setName would be a good name for a mutator method that sets a student's first and last name instance variables, and getLast would be a good name for an accessor method that returns the last name.

Braces and Indentations

As shown in Figure 8.1 and Figures 8.2a and 8.2b, place opening braces ({) immediately below the first letter of the preceding line. Indent everything that's logically inside the brace. When you're done with a block (that is, when you're ready for the closing brace), "outdent" so the opening and closing braces for a particular block are aligned. By following this indent-outdent scheme, you'll always align opening and closing brace partners in the same column. For example, note how the Student class's opening and closing braces are both in the same column.

Our recommendation on where to put the opening brace ({) is different from Oracle's recommendation, which is that the opening brace be at the end of the previous line, like this:

```
public void setName(String first, String last) {
  this.first = first;
  this.last = last;
}
```

This is one of the few places where our recommendation differs from Oracle's recommendation. Many programmers follow the recommendation that we prefer because it provides better visual bracketing of the block of code that the braces define. However, placing the opening brace at the end of the previous line makes the code a little tighter, and if you or your teacher or your boss wants the opening brace at the end of the previous line, you have our blessing to follow that convention.

Be consistent with your indentations. Any indentation width between two and five spaces is acceptable as long as you're consistent throughout your program. We use two spaces in the book because book page widths are less than computer screen widths, and we don’t want to run out of room for programs with deep nesting.

⚠ Many novice programmers indent improperly. They either don’t indent when they should indent, or they indent when they shouldn’t indent, or they use inconsistent indentation widths. That leads to programs that are unprofessional-looking and difficult to read. Some novice programmers postpone entering their indents until the end, after they’ve finished debugging. Big mistake! Use proper indentation as you enter your program. That should be pretty easy since there are really only two rules to remember:

1. Use braces to surround a block of code that is logically inside something else.
2. Indent the code that is inside the braces.

There is one exception to the first rule:

> Code after the `->` in a switch construct's case clause is considered to be logically inside the case clause, but braces are not used (unless the case clause has multiple statements or has one or more statements before a final expression).

Variable Declarations

As shown in Figure 8.1's main method, place all local variable declarations at the top of the method (even though that's not required by the compiler). Exception: Unless you need a for loop iteration variable to persist beyond the end of the for loop, declare it in the initialization field of the for loop header.

Normally, specify only one variable declaration per line. Exception: If several variables with obvious meanings are intimately related, it's OK to group them on one line.

Include a comment for every variable whose meaning is not obvious. For example, the cryptic local variable declarations in the main method in Figure 8.1 definitely need comments, and we also provide comments for the instance-variable declarations in Figure 8.2a. Note how those comments are aligned—their //'s are in the same column. In general, if you have comments that appear at the right side of several nearby lines, try to align those comments.

Line Wrap

If you have a statement that is too long to fit on one line, split it at one or more natural breaking points within the statement. For example, note where we break the long print statement in Figure 8.2b's setFirst and setLast methods. We consider these to be natural breaking points:

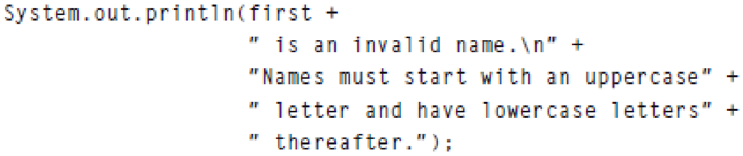
- right after the opening

parenthesis

- after a concatenation operator
- after a comma that separates parameters
- at whitespace in expressions

After a break point in a long statement, indent the remaining part of the statement on the next line. In - Figure 8.2b, note how we indented the continuation lines with the same standard two-space width that we use for all other indentations.

Rather than simply indenting continuation lines with the standard indentation width, some programmers prefer to align continuation lines with a parallel entity on the previous line. For example, in the aforementioned print statement, they would align the continuation line with first like this:



```
System.out.println(first +
                   " is an invalid name.\n" +
                   "Names must start with an uppercase" +
                   " letter and have lowercase letters" +
                   " thereafter.");
```

In our opinion, the above code is pushed too far to the right and is unnecessarily chopped up. That's why we prefer to keep it simple and just indent with the normal indentation width.

Braces That Surround One Statement

For a loop statement or an if statement that includes only one subordinate, it's legal to omit the braces around the statement. For example, in Figure 8.2b's setFirst method, the "if-else" statement could be written like this:

```
if (first.matches("[A-Z][a-z]*"))
  this.first = first;
else
  System.out.println(first + " is an invalid name.\n" +
    "Names must start with an uppercase letter and have" +
    " lowercase letters thereafter.");
```

However, we like to use braces for all loop statements and if statements, even if there is only one enclosed statement. Why?

- Braces provide a visual cue for remembering to indent.
- Braces help you avoid a logical mistake if you add code later that's supposed to be within the loop statement or the if statement.

The second point can best be understood with an example. Assume that a program contains this code:

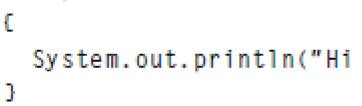
```
if (person1.isFriendly())
  System.out.println("Hi there!");
```

Assume that a programmer wants to add a second print statement ("How are you?") for a friendly person1 object. A careless programmer might do it like this:

```
if (person1.isFriendly())
  System.out.println("Hi there!");
  System.out.println("How are you?");
```

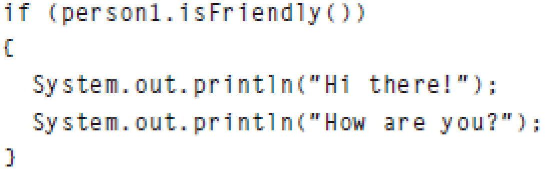
Because the second print statement is not within braces, it is executed regardless of whether person1 is friendly. And do you want to ask an unfriendly person "How are you?" You might get a scowl for a response. ⚠

On the other hand, if the program followed our style guidelines, the original code would look like this:



```
if (person1.isFriendly())
{
  System.out.println("Hi there!");
}
```

Then if a programmer wants to add a second print statement (“How are you?”) for a friendly person1 object, it would be harder to make a mistake. Even a careless programmer would probably code the second print statement correctly like this:



```
if (person1.isFriendly())
{
  System.out.println("Hi there!");
  System.out.println("How are you?");
}
```

In our above discussion, we said that “we like to use braces for all loop statements and if statements.” More formally stated, we like to use a block for all loop statements and if statements. A *block* is a set of statements surrounded by braces.

Comments

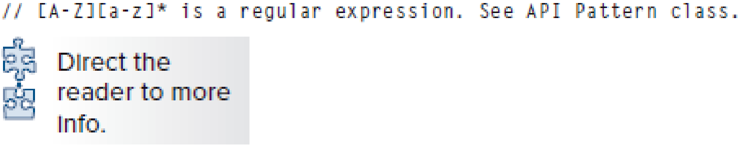
As shown in Figure 8.1 and Figures 8.2a and 8.2b, for all but the shortest blocks, include a comment after a closing brace in order to specify the block that is being closed. For example, in Figure 8.2b, note this closing brace line for the setFirst method:

```
} // end setFirst
```

Why is that good practice? So someone reading the program can quickly identify the block that is being ended without having to scroll to the top of the block to find out. It’s OK to omit closing-curly-brace comments for short blocks of less than about five lines. For short blocks, it’s easy to tell what block the closing brace is attached to, and the final comment just adds clutter.

Include comments for code segments that would not be obvious to a typical

Java programmer. In Figure 8.2b, notice this comment that appears at the tops of the bodies of the setFirst and setLast methods:



```
// [A-Z][a-z]* is a regular expression. See API Pattern class.
```

This comment is helpful because the subsequent statement is more obscure than most. A comment should either explain something directly, help the programmer find more information on a topic, or both. A comment like the one above that references an authoritative source is especially important whenever code implements something mysterious—an arbitrary definition like the "regular expression" above, a formula with empirical coefficients, or a mysterious mathematical expression.

Whenever a comment is too long to fit at the right of the line that is being explained, put it on one or more lines by itself above the line that is being explained. The // should be indented the same as the described line. If you put a comment on a line by itself, make sure there is sufficient whitespace above it. In the setFirst and setLast methods of Figure 8.2b, there’s sufficient whitespace above the comments because the prior lines happen to be opening braces for their respective method bodies. In other cases, you’ll need to insert a full blank line above the comment. It’s optional whether you insert a blank line below it.

Do not add individual comments that just restate what the code already tells you. For example, for the first assignment statement in Figure 8.1’s main method, this comment would be overkill:

```
s1 = new Student(); // instantiate a Student object
```

Developing readable programs is an important skill and a bit of an art form.

Having too few comments is bad because it leads to programs that are difficult to understand. But having too many comments is also bad because it leads to cluttered programs that are difficult to wade through. There’s a similar balancing act for blank lines. Having too few blank lines is bad because it leads to programs that are difficult to understand; but having too many blank lines is also bad because it leads to programs with too much dead space.

Blank Spaces

As shown in Figure 8.1 and Figures 8.2a and 8.2b, include blank spaces:

- after the single asterisks in the prologue
- before and after all operators (except for the operators inside a for loop header)
- between a closing brace and the

//’s for its associated comment

- after the //’s for all comments
- after the if, while, and switch keywords

On the other hand, do not include blank spaces:

- between a method call and its opening parenthesis
- within each of the three components in a for loop header

The last point can best be understood with an example. Here is a nicely written for loop header:

for (int i=0; i<10; i++)

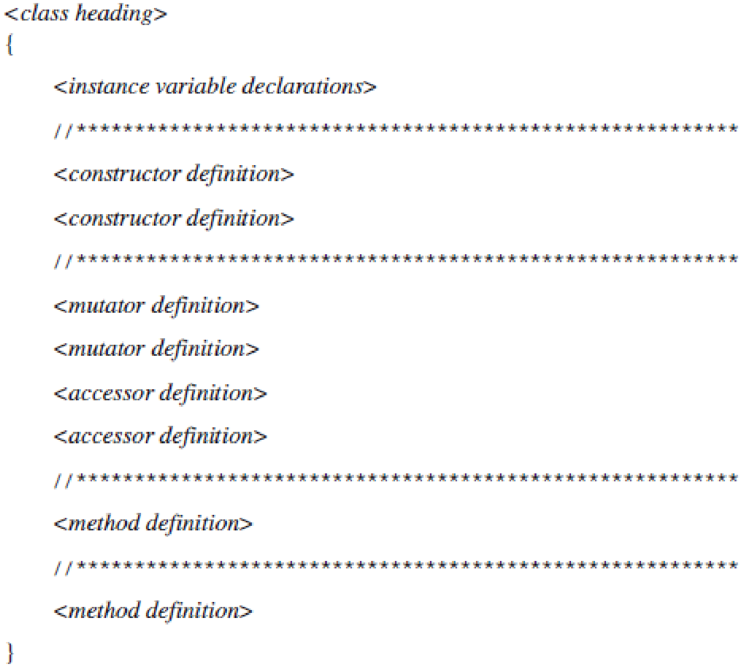
Note that there are no spaces surrounding the = operator or the < operator. Why is that good practice? Because the for loop header is inherently complex. In order to temper that complexity, we add visual cues to compartmentalize the for loop header.

More specifically, we consolidate each section (no spaces within each section), and we insert a space after each semicolon to keep the three sections separate.

Grouping Constructors, Mutators, and Accessors

For short, obvious methods, you should omit descriptions. For example, mutators and accessors are short and obvious, so you should omit descriptions for them. Constructors are sometimes short and obvious, but not always. If a constructor simply assigns parameter values to associated instance variables, then it is short and obvious and you should omit a description for it. If, on the other hand, a constructor performs nonobvious input validation on user-entered values prior to assigning them into associated instance variables, then you should include a description for the constructor.

In the interest of grouping similar things together, we recommend omitting the line of asterisks between mutators and accessors and between short obvious constructors. Assuming that a class contains two short, obvious constructors, several mutator and accessor methods, and two short, obvious other methods, here’s the framework for such a class:



```
<class heading>
{
    <instance variable declarations>
    //***********************************************************
    <constructor definition>
    <constructor definition>
    //***********************************************************
    <mutator definition>
    <mutator definition>
    <accessor definition>
    <accessor definition>
    //***********************************************************
    <method definition>
    //***********************************************************
    <method definition>
}
```

For this case, there are no descriptions for the constructors, the accessors, or the mutators. There is a line of asterisks above the firstmutator, but not above the subsequent mutator and accessors. These omissions make the program more readable by grouping similar things together.

8.3 Documentation for Outsiders

The preceding section described coding strategies that make code easier to understand, so the reader will be better able to know how to edit or improve it, if necessary. This section also describes techniques that make code easier to understand, but this time the goal is for the reader to be able to use the code (not to edit or improve it). As you might expect, there is some overlap, but that’s OK. The previous section’s style is right for one audience. That was documentation for insiders. This section’s style is right for a different audience. This is documentation for outsiders.

Preconditions and Postconditions

Suppose you are thinking about using an existing block of code like a class or a method, but you can’t or don’t want to dive into the details of how it works. You’ll still need to know what it does. Well-written Java code is largely self-documenting. If a program is written with full, descriptive words for its class names and method names, that can help someone using the program discern what the program does.

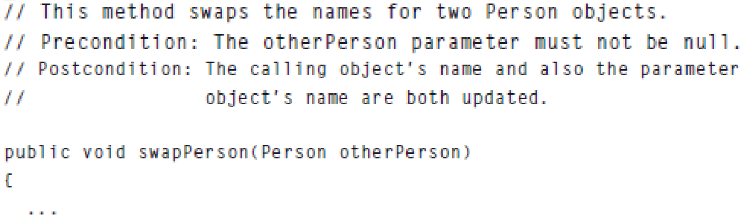
Comments in a class’s prologue might address either what the class does or how it does it. Thus, comments communicate with outsiders as well as insiders. Comments just above a method header might describe what a method does as much as how it does it. So they also communicate with outsiders as well as insiders. Up to now, we've been flexible about the content of those comments. Now, let's move more to the left sides of our brains. Let's think more like lawyers and try to convert some of the comments above our methods into legal *contracts*.

We can make two kinds of contracts. One kind of contract, called a *precondition*, is an assumption. It says the method expects a certain condition or set of conditions to be true before it executes. If the precondition is not satisfied, then the results produced by the method are unpredictable. The other kind of contract, called a *postcondition*, is a guarantee or warranty. It says that the method promises to make certain conditions true by the time it finishes execution. A method's preconditions are what it needs. Its postconditions are what it does.

For many methods, like ordinary accessor and mutator methods, the method and parameter names themselves provide all the necessary precondition and postcondition information. In other cases, an ordinary comment above the method header provides enough additional information to make it clear what the preconditions and postconditions are. Whenever a comment describes the state of a parameter or the state of an instance or static variable that the method relies on, that's an example of providing precondition information. Similarly, whenever a comment describes what a method returns or describes the state of an instance variable or static variable that the method updates, that's an example of providing postcondition information.

If you feel it's appropriate to

emphasize a particular precondition assumption or a particular postcondition guarantee, you should consider providing an explicit precondition comment or an explicit postcondition comment. The following example shows an updated version of Figure 7.7's swapPerson method, with such explicit precondition and postcondition comments.



```
// This method swaps the names for two Person objects.
// Precondition: The otherPerson parameter must not be null.
// Postcondition: The calling object's name and also the parameter
//                object's name are both updated.

public void swapPerson(Person otherPerson)
{
  ...
```

The precondition above is particularly important because if otherPerson is null, the program will compile successfully, but it will crash when executed. The postcondition above is also important because it alerts the method user of something fairly unusual. Normally, parameters transport information one way—from the calling module to the called method. But in this case, the otherPerson parameter does more. The method updates the value in otherPerson's name instance variable, and that updated value is accessible to the calling module.

Starting now, from time to time we will provide explicit precondition and/or postcondition comments to emphasize important conditions in a method. However, to avoid overburdening you, we usually try to avoid situations that require such comments, so you will not see many of them.

Introduction to javadoc

Now let's move further from "how" to "what"—to what we see in Java's application programming interface (API) documentation, where the "how" is completely hidden. Did you ever wonder how they produce that documentation? Do people familiar with Java create web pages from information in their heads, or does a computer program generate those web pages automatically? A computer program generates them automatically. That program is the javadoc tool, which (along with javac and java) is an integral part of Oracle's Java Development Kit (JDK).

Like a word processor, javadoc copies certain patterns in Java source code and pastes them onto web pages. The designers of Java API software use javadoc to document the code they write. We too can use javadoc to document the code we write. To do this, enter this command at a command prompt:

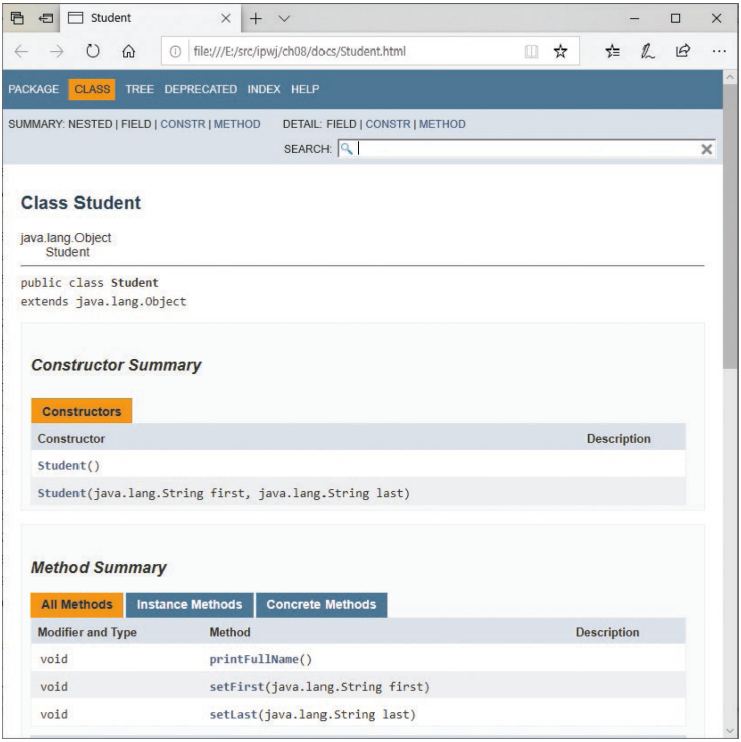
```
javadoc -d output-directory source-files
```

The -d *output-directory* option[3] ("d" means "destination") causes the output to go to the specified output directory. You can put documentation for more than one source code file into the same documentation directory. Just put spaces between multiple source file names.

Suppose you want to generate interface documentation on the Student class whose source code is presented in Figures 8.2a and 8.2b. Assuming you are currently in the directory that contains the source code, and assuming you want javadoc's output to go to a subdirectory called docs, here's what the command would look like:

```
javadoc -d docs Student.java
```

To see the web page javadoc creates for Student.java, navigate to its docs file, and open the file called Student.html. Figure 8.3 shows the top part of the web page that javadoc creates—the "Summary" information. This interface document contains an impressive amount of information—but not quite everything we need. For example, it doesn't include the comment in the last line of the prologue that describes the class in general, it doesn't include the comment that describes the two-parameter constructor, and it doesn't include the comments that describe the three methods.


Student
file:///E:/src/ipwj/ch08/docs/Student.html
PACKAGE
CLASS
TREE
DEPRECATED
INDEX
HELP
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH:

Class Student

java.lang.Object
    Student

```
public class Student
extends java.lang.Object
```

Constructor Summary

Constructors

Constructor	Description
`Student()`	
`Student(java.lang.String first, java.lang.String last)`	

Method Summary

All Methods | Instance Methods | Concrete Methods

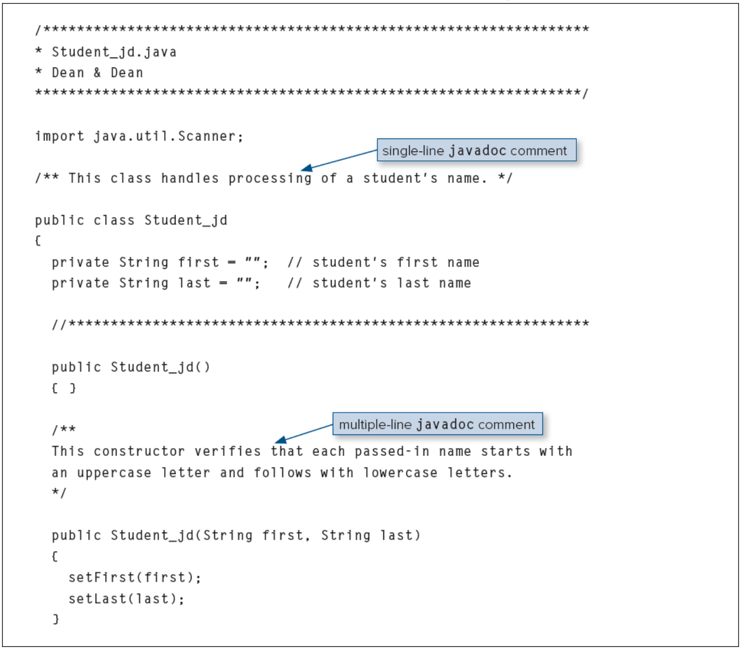
Modifier and Type	Method	Description
`void`	`printFullName()`	
`void`	`setFirst(java.lang.String first)`	
`void`	`setLast(java.lang.String last)`	

Figure 8.3 Top part of the javadoc output for the Student class defined in Figures 8.2a and 8.2b

To enable javadoc to extract this other information from source code, we need for all interface information to be located immediately above the heading of whatever it is describing. Also, we need for this information to be enclosed in a javadoc *block comment* that begins with a single forward slash followed by two asterisks and ends with a single asterisk followed by a single forward slash, like this:

> /** <*extractable information*> */

Since Figure 8.2a has an import statement between the general prologue and the class heading, we must move our general comment out of the general prologue and put it into a javadoc block comment located just above the class heading. Similarly, we must put individual constructor and method interface information into javadoc block comments located just above their respective headings. There is some flexibility. The extractable information in one of these javadoc block comments does not need to be on just one line. Also, if you wish, you may put the opening /** and the closing */ on lines above and below the text, as shown in Figure 8.4.



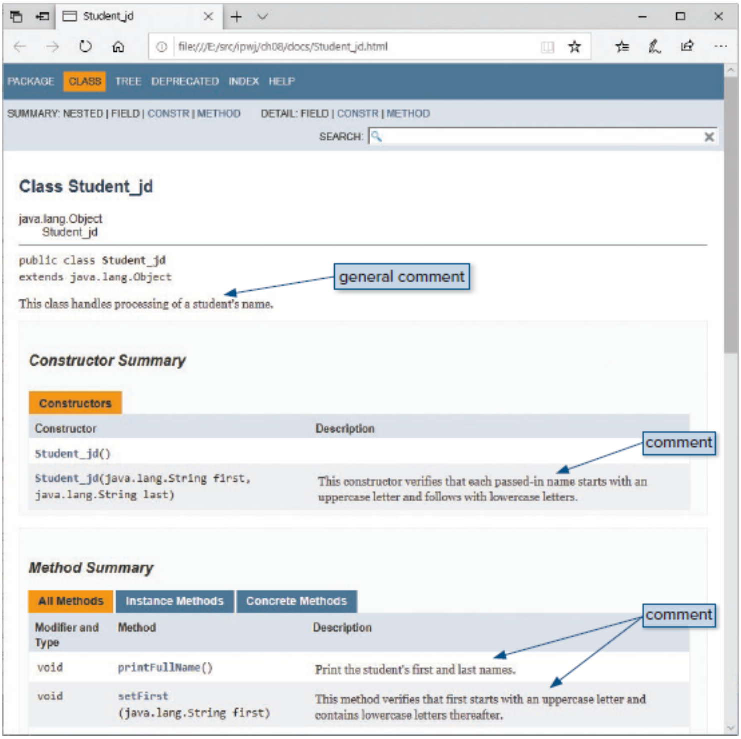
```
/******************************************************************
* Student_jd.java
* Dean & Dean
******************************************************************/

import java.util.Scanner;

/** This class handles processing of a student's name. */

public class Student_jd
{
  private String first = "";  // student's first name
  private String last = "";   // student's last name

  //**************************************************************

  public Student_jd()
  { }

  /**
  This constructor verifies that each passed-in name starts with
  an uppercase letter and follows with lowercase letters.
  */

  public Student_jd(String first, String last)
  {
    setFirst(first);
    setLast(last);
  }
```

Figure 8.4 Top part of the Student class in Figure 8.2a, modified to accommodate javadoc

With these changes implemented in the Student.java code, Figure 8.5 shows the top part of what javadoc generates. If you compare this with Figure 8.3, you'll see that Figure 8.5 includes the general comment for the whole class and the special comment for the two-parameter constructor. We also changed the rest of the code so that Student_jd has /** ... */ javadoc block comments above the method headings too. Therefore, the javadoc output also includes special comments for each method. The constructor and method comments also appear in "Detail" parts of the output display, which are below what you see in Figures 8.3 and 8.5.

This is just an introduction. For more details, see Appendix 6. In professional programming, it's important to use javadoc to document your code properly. However, as you can see in

Appendix 6, this makes code significantly longer. Textbook descriptions accompanying code presentations parallel much of the information in javadoc comments and make such comments redundant in the textbook context. Therefore (except for what you see in this section and in Appendix 6), to save book pages and lighten your load, this book's programs do not include the extra comments and apparatus needed for proper javadoc documentation.



Student_jd
file:///E:/src/ipwj/ch08/docs/Student_jd.html
PACKAGE
CLASS
TREE
DEPRECATED
INDEX
HELP
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
SEARCH:
general comment
comment
comment

Class Student_jd

java.lang.Object
    Student_jd

```
public class Student_jd
extends java.lang.Object
```

This class handles processing of a student's name.

Constructor Summary

Constructors

Constructor	Description
`Student_jd()`	
`Student_jd(java.lang.String first, java.lang.String last)`	This constructor verifies that each passed-in name starts with an uppercase letter and follows with lowercase letters.

Method Summary

All Methods
Instance Methods
Concrete Methods

Modifier and Type	Method	Description
`void`	`printFullName()`	Print the student's first and last names.
`void`	`setFirst(java.lang.String first)`	This method verifies that first starts with an uppercase letter and contains lowercase letters thereafter.

Figure 8.5 Top part of the javadoc output for the modified Student class

©Oracle/Java

8.4 Helper Methods

In the first four chapters, we solved essentially every problem we addressed in just one module—the main method. As problems get bigger, however, it becomes more and more necessary to partition them into subproblems, each of which has a manageable size. We started doing this in earlier chapters when our main method called on some of Java's API methods for help.

In a broad sense, any method that is called by another method is a "helper method" because the called method helps the calling method. The calling method is a client, and the called method (the broad-sense helper method is a server.

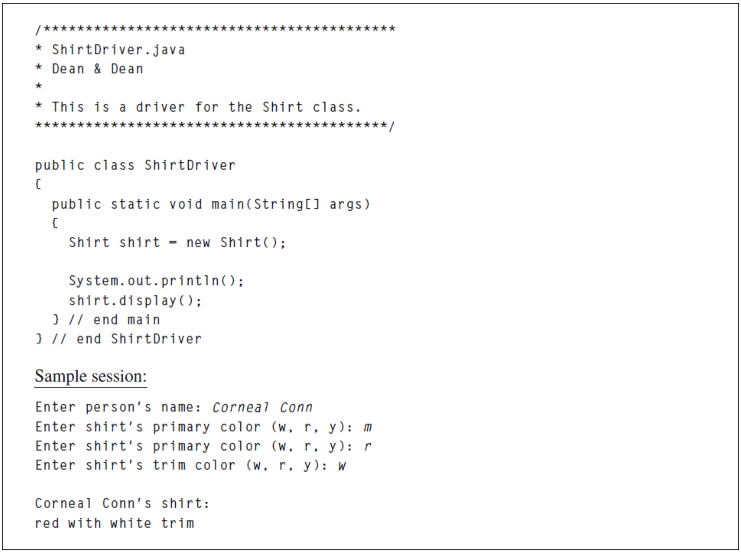
You can narrow the definition of helper method by restricting it to a called method that happens to be in the same class as the calling method. In Figure 8.2a, the Student constructor calls two methods from the Student class—setFirst and setLast. Presumably, these mutators were written to allow a user to change the instance variables in an object after the object was originally initialized. But once their code is written, why not reuse it? By including calls to these two ordinary methods in the constructor, we avoid duplication of the code in the called methods. Because the setFirst and setLast mutator methods each include a significant amount of error-checking code that helps the constructor do its job, this organization helps divide the problem into smaller chunks.

You can narrow the definition of helper method even more. Up to this point, all methods we've covered have used the public access modifier. These public methods are part of the class's *interface* because they are responsible for the communication between an object's data and the outside world. Sometimes you'll want to create a method that is not part of the interface; instead, it just supports the operation of other methods within its own class. This special type of method

—a method that is in the same class and has a private access modifier—is what we mean when we say *helper method.*

Shirt Program

Suppose you're asked to write a program that handles order entries for sports-uniform shirts. For each shirt order, the program should prompt the user for a shirt's primary color and its trim color. Figure 8.6 shows the driver and a typical sample session.



```
/*******************************************
* ShirtDriver.java
* Dean & Dean
*
* This is a driver for the Shirt class.
*******************************************/

public class ShirtDriver
{
  public static void main(String[] args)
  {
    Shirt shirt = new Shirt();

    System.out.println();
    shirt.display();
  } // end main
} // end ShirtDriver
```

Sample session:

```
Enter person's name: Corneal Conn
Enter shirt's primary color (w, r, y): m
Enter shirt's primary color (w, r, y): r
Enter shirt's trim color (w, r, y): w

Corneal Conn's shirt:
red with white trim
```

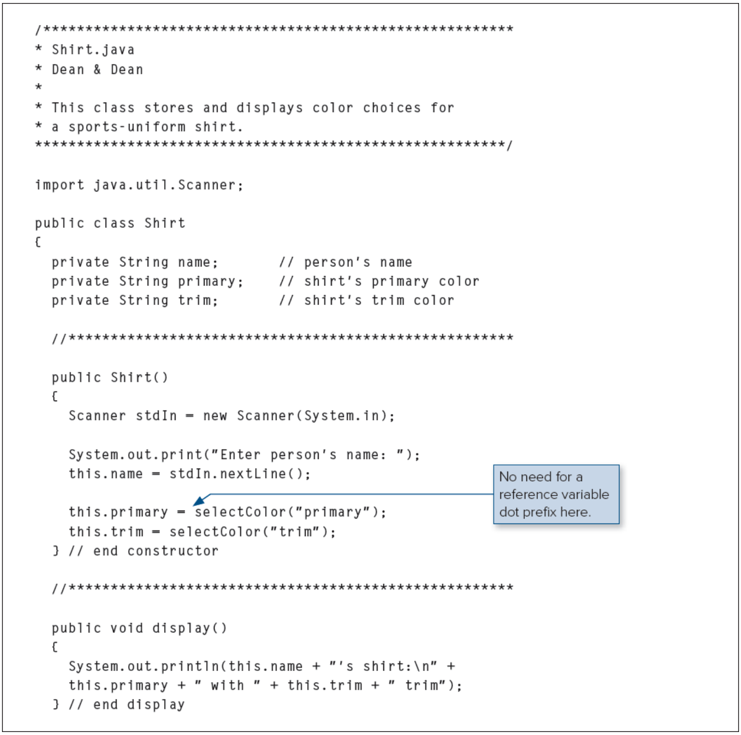
Figure 8.6 ShirtDriver class and

associated sample session

For each color selection, the program should perform the same input validation. It should verify that the entered color is one of three values—w, r, or y, for white, red, or yellow. That input validation code is nontrivial. It's in charge of:

- prompting the user for a color entry,
- checking whether the entry is valid,
- repeating the prompt if the entry is invalid, and
- converting the single-character color entry to a full-word color value.

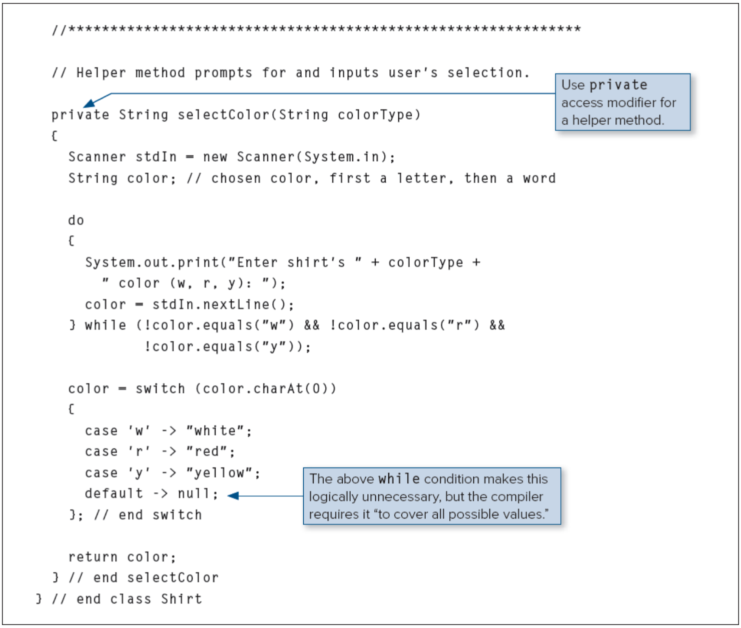
These four tasks are a coherent group of activities. Therefore, it's logical to encapsulate them (bundle them together) in a separate module. The fact that each shirt order performs this coherent group of activities two separate times (once for selecting the shirt's primary color and once for selecting the shirt's trim color) provides an additional reason to encapsulate them in a separate module. Thus, instead of repeating the complete code for these four tasks in the constructor each time color selection is needed, you should put this color-selection code in a separate helper method and then call that method whenever color selection is needed. Study the Shirt program and sample session in Figures 8.6, 8.7a, and 8.7b, especially the public constructor, Shirt, and the private helper method, selectColor. Note how the constructor calls the selectColor method twice. In this particular case, the helper method calls are from a constructor. You can also call a helper method from any ordinary method in the same class.



```
/*******************************************************
* Shirt.java
* Dean & Dean
*
* This class stores and displays color choices for
* a sports-uniform shirt.
*******************************************************/

import java.util.Scanner;

public class Shirt
{
  private String name;       // person’s name
  private String primary;    // shirt’s primary color
  private String trim;       // shirt’s trim color

  //*****************************************************

  public Shirt()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print(”Enter person’s name: ”);
    this.name = stdIn.nextLine();

    this.primary = selectColor(”primary”);
    this.trim = selectColor(”trim”);
  } // end constructor

  //*****************************************************

  public void display()
  {
    System.out.println(this.name + ”’s shirt:\n” +
    this.primary + ” with ” + this.trim + ” trim”);
  } // end display
```

No need for a reference variable dot prefix here.

Figure 8.7a Shirt class—part A



```
//***************************************************************

// Helper method prompts for and inputs user's selection.

private String selectColor(String colorType)
{
  Scanner stdIn = new Scanner(System.in);
  String color; // chosen color, first a letter, then a word

  do
  {
    System.out.print("Enter shirt's " + colorType +
      " color (w, r, y): ");
    color = stdIn.nextLine();
  } while (!color.equals("w") && !color.equals("r") &&
           !color.equals("y"));

  color = switch (color.charAt(0))
  {
    case 'w' -> "white";
    case 'r' -> "red";
    case 'y' -> "yellow";
    default -> null;
  }; // end switch

  return color;
} // end selectColor
} // end class Shirt
```

Use `private` access modifier for a helper method.

The above `while` condition makes this logically unnecessary, but the compiler requires it "to cover all possible values."

Figure 8.7b Shirt class—part B

Page 341

There are two main benefits to using helper methods: (1) By moving some of the details from public methods into private methods, they enable the public methods to be more streamlined. That leads to public methods whose basic functionality is more apparent. And that in turn leads to improved program readability. (2) Using helper methods can reduce code redundancy. Why is that? Assume that a particular task (such as color input validation) needs to be performed at several places within a program. With a helper method, the task's code appears only once in the program, and whenever the task needs to be performed, the helper method is called. On the other hand, without helper methods, whenever the task needs to be performed, the task's complete code needs to be repeated each time the task is done.

Note that in Figure 8.7a, we call the selectColor method without a reference variable prefix:

```
this.primary = selectColor("primary");
```

Why is there no reference variable dot prefix? If you're in a constructor (or an instance method, for that matter), and you want the current object to call another method that's in the same class, the reference variable dot prefix is unnecessary. Because the constructor and the selectColor method are in the same class, the reference variable dot prefix is unnecessary.

8.5 Encapsulation (with Instance Variables and Local Variables)

A program exhibits encapsulation if its data is hidden—that is, if its data is difficult to access from the “outside world.” Why is encapsulation a good thing? Because the outside world isn’t able to access the encapsulated data directly, it’s more difficult for the outside world to mess things up.

Encapsulation Implementation Guidelines

There are two main techniques for

implementing encapsulation:

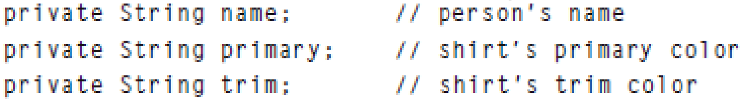
- First, break a big problem into separate classes, where each class defines a set of encapsulated data that describe the current state of an object of that class. Encapsulate this object-state data by using the -private access modifier for each such data item. As you already know, a class's object-state data items are called *instance variables*.
- Second, break a class's tasks into separate methods, where each method holds a set of additional encapsulated data it needs to do its job. As you already know, a method's data items are called *local variables.*

Declaring instance variables within a class is one form of encapsulation, and declaring local variables within a method is another form of encapsulation. Which is the stronger (more hidden) form of encapsulation? All instance methods have access to all instance variables defined in the same class. On the other hand, only the current method has access to one of its local variables. Therefore, a local variable is more encapsulated than an instance variable. Thus, to promote encapsulation, use local variables, not instance variables, whenever possible.

⚠ In writing a method, you'll often find the need for more data than what's provided by the current instance variables. The question then becomes—how should you store that data? In another instance variable? Or locally? Try to resist the urge to add another instance variable. You should use instance variables only for storing fundamental attributes of the class's objects, not for storing additional details. If you can store the data locally, then do so. That furthers the goal of encapsulation. Usually when we think of storing data locally, we think of a local variable declared inside a method's body. Be aware that parameters are another way to store data locally. Remember that a parameter is declared in a method's heading—that tells us it has local scope.

Local Variables Versus Instance Variables in The Shirt Class

Now let's see how the above philosophy plays out in the Shirt class. The fundamental attributes of a shirt are its name, its primary color and its trim color. That's the basis for our declaration of the three instance variables declared in Figure 8.7a:



```
private String name;        // person's name
private String primary;     // shirt's primary color
private String trim;        // shirt's trim color
```

Now let's look at the other variables we need as we write the class's methods. All of these other variables are somehow associated with the selectColor method in Figure 8.7b. We need to transfer data in both directions between the calling Shirt constructor and the called selectColor method.

First, consider transfer of data into the selectColor method. If a shirt’s primary color is needed, then selectColor should print this prompt message:

Enter shirt's primary color (w, r, b):

If a shirt’s trim color is needed, then selectColor should print this prompt message:

Enter shirt's trim color (w, r, b):

We must transfer data into the selectColor method that tells the selectColor method which query to print. It would be possible to transfer this data by declaring another instance variable called colorType, have the Shirt constructor write a value to this instance variable, and then have the selectorColor method read the value of this instance variable. But this would be bad practice because it would break the encapsulation within the selectColor method and add confusing clutter to our nice clean list of object attributes. The proper way to implement this method-to-method communication is the way we did it, with an argument/parameter transfer.

⚠

Second, consider transfer of data out of the selectColor method. We also have to transfer data back from the selectColor method to the Shirt constructor. This data is the string representation of the selected color. There are three good ways to transfer data back to the calling method:

1. If there is only a single return value, you can send it back to the calling module as a return value.
2. If there is more than one value to return, you can assemble these

values into an object, create that object in the helper method, and return a reference to that locally created “communication object.”

3. You can pass into the helper method references to “communication objects” instantiated in the calling module and use code in the helper method to write to those objects.

It’s also possible to transfer data back to the calling module by declaring other instance variables, having the helper method write values to them, and having the calling module read from them after the helper method terminates its execution. But this would be bad practice because it would break the encapsulation and add confusing clutter to our nice clean list of object attributes. The proper way to implement this method-to-method communication is the way we did it, with a return value. In this case, the return value is a reference to a String object. ⚠

The Shirt class has one other variable to consider, the stdIn reference to a keyboard communication object. This particular object is used by both the calling constructor and the called helper method, and it is instantiated twice, once in each of those two modules. It is tempting to try to avoid duplicate instantiation by making stdIn an instance variable. And it will “work.” But we recommend against doing this because stdIn is clearly not a fundamental attribute of this class’s objects. It’s not a variable that describes the state of a shirt! In a later version of the program, you might want to change the method of input from the keyboard to something else, like a data file, described later in Chapter 16. You might even want to use one method of input for the name and a different method of input for the other state variables. Then you'd need to change stdIn, and you might want to change it in different ways for different methods. Declaring it local makes future modifications local also, and it's better design practice.

An argument used for not making a variable local is “maybe someday we'll need broader scope.” If you have a specific plan that truly requires the broader scope you propose, OK. But if it's just “maybe someday,” don't provide broader scope until that “someday” actually comes. Then, at that time, modify your program to increase scope only where it's absolutely necessary. ⚠

8.6 Recognizing the User's Point of View

Suppose all you want to do is use somebody else's program, and you don't really care how it's written. In that case, you are a user, and what you want is a user's view of the program. Every time we present a "sample session" or a GUI display, we're showing you what users see.

Sometimes the intended users are other programmers. Then, the most appropriate user's view is documentation designed to be read by programmers. An excellent example of this kind of user's view is the Java API documentation or the output produced by javadoc, described in Section 8.3 and Appendix 6. In Chapter 5, you looked at the interfaces of code written by designers of the Java language. We were not showing you how to write that code; we were just showing you how to use it. That view of the Java API was a user's view for users who are themselves programmers.

Sometimes the intended users may not be programmers. They might be artists, accountants, nurses or doctors, detectives, librarians, inventory clerks, or equipment service personnel. Sometimes whole books are devoted to explaining a big program to users who are not programmers. The goal of these books is not to explain how to write the program, or even how it works. It is to explain what the program does and how to use it to solve a problem that is outside the world of programming. Figure 8.8 shows the principal components of program documentation written for users who may not be programmers.

When you write a program that someone else will use, occasionally you should try to put yourself into that other person's "shoes." If you know some users or potential users, visit with them. Ask them what their needs are, and try to determine how they like to think about the subject the program will address. If possible, get them to make a list or draw a picture that portrays the results they would like the program to produce. Ask what kinds of data they would like the program to use as input and what kinds of assumptions they consider to be acceptable.

Try to think about the problem the same way users are inclined and able to think about the problem. Ask them how they have solved the problem or similar problems in the past. When numerical calculations are involved, ask how they have done those calculations. If potential customers are in the market for a computer solution, they will expect greater accuracy and speed from the computer, and they will expect your solution to be better than theirs. But if you know what they have done in the past, you will be more able to give them something they will like, and you will be more prepared for their evaluation of your product.

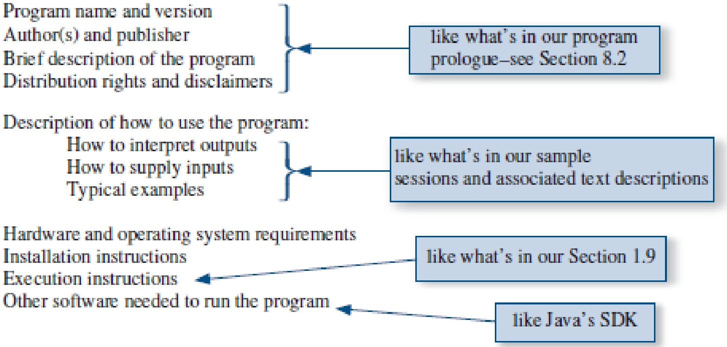

Program name and version
Author(s) and publisher
Brief description of the program
Distribution rights and disclaimers
like what's in our program prologue–see Section 8.2
Description of how to use the program:
How to interpret outputs
How to supply inputs
Typical examples
like what's in our sample sessions and associated text descriptions
Hardware and operating system requirements
Installation instructions
Execution instructions
Other software needed to run the program
like what's in our Section 1.9
like Java's SDK

Figure 8.8 Principal components of documentation for users who may not be programmers

As you develop the program, keep thinking about output and input formats—output first, then input. From time to time, test your proposed formats with potential users. When you do this, listen for feedback that goes beyond cosmetics. Are you focusing on the problem users care about? Are your assumptions the same as theirs? Have you left out anything that's important? Learning how to think like a user will improve your effectiveness as a programmer.

8.7 Design Philosophy

In the next several sections, we discuss alternative strategies for solving problems. That's plural "strategies" because there's not just one cookie-cutter strategy that can be used to solve all problems. If there were just one universal strategy, programming would be easy and anyone could do it. But it's not easy. That's why good programmers are in demand and earn a decent wage.

Simplistic Approach to Design

Here's a simplistic recipe for how to design things:

1. Figure out what you want to do.
2. Figure out how to do it.
3. Do it.

4. Test it.

At first, this list seems like obvious common sense. But actually, it works only for very simple problems—problems where everything is easy and you don’t need any recipe. What’s wrong with this recipe?

First, if a problem is difficult, it’s hard to know what its solution will be like. Often we need experience to know even what we want to do. Most clients recognize this and are flexible enough to accept a range of possible solutions. They want to avoid imposing arbitrary specifications that would cause them to miss inexpensive opportunities or incur expensive penalties. With difficult problems, people want to keep their options open.

Second, most problems have several alternate ways in which they can be solved. It takes some experimentation to determine the best way to solve a difficult problem. For very difficult problems, it’s impossible to know exactly “how to do it” until we have done it.

Third, when we “do it,” we must recognize it will not be perfect. There will be hidden errors. We will discover a better way to do it. The client will discover it would have been better to have asked for something different. And we’ll need to do it again.

Fourth, if we defer testing of anything complicated until the end, we are almost sure to fail. The thing might pass its one final “test,” but it will probably fail in its ultimate job, because one final test cannot catch all problems.

So, how can you deal with these difficulties?

1. Develop and maintain a sensible compromise between tight specification and flexibility.
2. Partition the problem into

distinct concerns.

3. Perform continuous testing at all levels. This helps you identify problems early when they are easy to fix, and it gives you objective assessment of progress. Suppose you're in charge of a large programming project, and you ask your programmers, "How's it coming?" You don't want them just to say, "fine." You want them to show you—by running tests that demonstrate what their current code actually does.

Separation of Concerns

When faced with a large problem, if there is time to reflect, humans like to partition the problem into manageable components. In programming, there's a formal term for the idea of putting code for each task into its own separate place—*separation of concerns*. More formally, we say that separation of concerns is when a program's functional behaviors are implemented in separate modules, with minimal overlap between the functional behaviors.

There are different ways to achieve separation of concerns within a program. At the most granular level, you can break up your program into methods, where each method performs one task. With OOP languages like Java, you can use classes to implement larger concerns, where each class implements its concern by storing related data and providing related methods. With many programming languages, if your program has even still larger concerns, you can keep the concerns separate by grouping a concern's classes together. Java implements that sort of grouping by grouping a concern's classes (where each class is a file) into its own package.

Design Patterns

The strategy of separating a program's concerns is so prevalent that software designers have agreed on separation-of-concern patterns that are particularly useful and popular. Such patterns are examples of *design patterns*, which are general-purpose software solutions to common problems. The software solutions can range in form from high-level descriptions to fully operational source code programs that can be used as starting-point templates. As a programmer, you should take the time to review some of the common design patterns so you know what's out there. That way, when you're confronted with a large, nontrivial problem, before you start implementing it from scratch, you'll have a sense of whether you can borrow from someone else's design. This can help you to avoid reinventing the wheel.

If you'd like to learn about common design patterns, a good place to start is Wikipedia's design pattern page. It describes quite a few design patterns, but we'll focus primarily on just one—the *model-view-controller*, or *MVC* for short. The MVC design pattern has been around a long time—so long that it actually predates the notion of a design pattern. Since it is such a useful design pattern, many integrated development environments include the MVC design pattern as a template that developers can generate with a few button clicks.

Here is how Microsoft describes the three parts (concerns) of the MVC design pattern.4

- **Model**. The model manages the behavior and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).

- **View**. The view manages the display of information.
- **Controller**. The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

Most of the real work is done by the model. The model stores the program's data (often in a database) and the program's rules (often referred to as *business logic*) that manipulate the data. The view presents the results to the user. Normally, the results are displayed with GUI using windows, text, pictures, etc. The controller is in charge of (1) gathering user input and (2) using that input to tell the model and the view what to do. The model can also tell the view what to do, but typically, it does so indirectly. When the model changes its state, if the view is set up to observe the model's state, the view then updates its presentation automatically. You'll see a simple example of the MVC separation-of-concerns design pattern later in this book, in Section 11.10.

Another design pattern is the *Composite* pattern, which organizes objects into compositional trees. We'll describe composition in detail later in this book, in Chapter 13. Another design pattern is the *Observer* pattern, which has objects "broadcast" their state changes to all interested observers.

Testing

It's been said that, on average, experienced programmers make one mistake for every 8 or 10 lines of code.[5] Whew! That’s a lot of mistakes. With such a high incidence of mistakes, we hope you’re properly convinced about the importance of testing.

Testing has three aspects:

Check the most obvious things first.

- First, subject your program to typical input values. If your program doesn’t work with typical input values, you’re in real trouble. Coworkers and users may question your competence if your program generates wrong answers for the typical cases.
- Second, subject your program to input values that are at the boundaries of acceptability. These boundary tests often reveal subtle problems that wouldn’t show up until later, and such problems might be much harder to fix at that time.

- Third, subject your program to invalid input values. In response to an invalid input value, your program should print a user-friendly message that identifies the problem and prompts the user to try again.

Testing is something that many people envision occurring after a product is finished. That’s an unfortunate notion, because a lone test at the end of the production of a complicated product is almost worthless. If the product fails such a test, it may be hard to determine why it failed. If the fix requires many changes, a great deal of work may have been wasted. If the product does not fail a lone final test, you may be lulled into thinking everything is OK even when it’s not. Passing a lone final test may actually be worse than failing a lone final test, because passing motivates you to release the product. It’s much more costly to fix a problem after a product has been released. (Ray knows about this!) Bottom line—Don't wait until the end to start your testing. Test your program on a regular basis throughout the development process. ⚠

Novice programmers sometimes get the idea that it would be "unscientific" to form a preconception of what a test result should be before you do the test. That's wrong. It's important that you do have a good idea of what the test result should be before you perform a test. Before you push the "run" button, say out loud what you think the result should be! This improves your chance of recognizing an error.

Testing keeps you on track. In any development program, you should interleave testing and coding so that you get quick feedback. If an experienced programmer makes a mistake in every 8 or 10 lines of code, a new programmer is well advised to perform some kind of test after every 4 or 5 lines of new code! This makes it easy to identify errors, and it reduces your level of stress. The more frequently you test, the more positive feedback you get, and this helps your attitude—it gives you a "warm-fuzzy feeling." Frequent testing makes programming a more pleasant experience.

There is no practical way to verify all the aspects of a complicated system by looking at it only from the outside. Testing should be performed on each component and on combinations of components, at all levels. As you'll see in subsequent discussions, testing typically requires creation of some kind of extra testing code. Sometimes it's a special driver. Sometimes it's a special driven module. Creating such test code may seem like extra work for something that's to be used only in a test environment and not in an actual runtime environment. Yes, it is extra work, but it is well worth the effort. Writing and using test code will save you time in the long run, and it will lead to a better final product.

Using assert Statements

Earlier in this chapter, we described preconditions and postconditions. They are examples of a more general construct—an *assertion*. Preconditions and postconditions represent an informal contract with outsiders who may wish to use your code but are not concerned with its details. Java also supports another kind of assertion—an assertion for insiders—that helps programmers debug code during development or modification. This assertion for insiders is a statement that begins with the Java keyword assert. It comes in either of two forms:

`assert` *Boolean-condition*`;`

or:

`assert` *Boolean-condition* `:` *string-describing-an-error*`;`

If its Boolean condition is true, an assert statement does nothing. If its condition is false, it prints an output saying an error has occurred.6 With the second form above, that output includes the descriptive string after the colon. A Java assert statement is like an if statement that prints an error message when the if condition is true. But there are three differences: (1) A true condition in an assert statement corresponds to a false condition in the corresponding if *(condition) then print* statement. (2) The assert statement terminates execution. (3) All assert statements are ignored unless you explicitly enable them. You can do that by running Java from a command line with the “enable assert” option (ea) specified like this:

java -ea *class-name-of-compiled-*

program

As an alternative, if you're using an integrated development environment to run Java, search for its enable assert configuration option.

It's common practice to use extra print statements to identify errors during program development. Then, after the bugs are gone, you remove those extra print statements to reduce code clutter and improve execution performance. If you use assert statements instead, you can leave the assert statements in the final code. The clutter they add is minimal, and when you execute without the -ea option, there is no performance penalty. If you need to modify the program later, you don't need to reintroduce the original extra print statements to verify the modifications. You can just reuse the still-present assert statements by once again executing with the -ea option.

To use assert statements effectively,

look for places in the code where you can ask true/false questions and get consistent and meaningful answers. One logical place is immediately after receiving incoming data. For example, to see if the precondition described above Section 8.3's swapPerson method header is satisfied, you might include this as the first statement in swapPerson's body:

```
assert otherPerson != null;
```

Assuming that you run the program with the -ea option, when the assert statement executes, the JVM checks to see if its condition (otherPerson != null) is true or false. If the condition is true, execution continues normally. If the condition is false, the JVM prints an error message and stops the program.

Another logical place is immediately after a series of related operations. For example, look back at the selectColor method in Figure 8.7b. The top half of this method has a loop whose -continuation condition looks at the first characters of the names of three particular colors. The bottom half of this method has a switch expression whose case clauses are supposed to be those same three characters. Is there consistency between the characters in the code on the top and the code on the bottom? We can enable the code on the bottom to check for consistency by changing the default clause from:

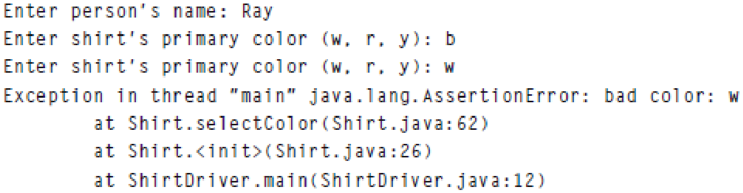
```
color = switch (color.charAt(0))
{
  case 'w' -> "white";
  case 'r' -> "red";
  case 'y' -> "yellow";
  default -> null;
}; // end switch
```

to:

```
color = switch (color.charAt(0))
{
  case ’w’ -> ”white”;
  case ’r’ -> ”red”;
  case ’y’ -> ”yellow”;
  default ->
  {
    assert false : ”bad color: ” + color.charAt(0);  //line 62
    yield null;
  }
}; // end switch
```

With a switch expression, if any case or default clause needs braces to accommodate a statement, everything else within such braces must be a statement too. The line beginning with assert is a statement, and the yield on the following line converts the null expression into a statement too.

If color specifications are consistent, the default clause never executes, and its assert statement can never execute. If color specifications are inconsistent, execution can sometimes proceed to the default clause. For example, suppose we comment out the switch expression’s first case 'w' -> "white"; Then we recompile and run the program with the -ea directive and input 'w' for the shirt’s primary color. The assert statement executes, and because its condition is always false, execution stops and the JVM prints an error message like what you see here after the w input:



```
Enter person's name: Ray
Enter shirt's primary color (w, r, y): b
Enter shirt's primary color (w, r, y): w
Exception in thread "main" java.lang.AssertionError: bad color: w
        at Shirt.selectColor(Shirt.java:62)
        at Shirt.<init>(Shirt.java:26)
        at ShirtDriver.main(ShirtDriver.java:12)
```

8.8 Top-Down Design

The dominant design methodology for large high-performance systems is the top-down design strategy. Top-down design requires the designer to think about the big picture first—that’s the “top.” After completing the design at the top, the designer works on the design at the next lower level. The design process continues in this iterative manner until the bottom level (the level with the most detail) is reached.

For an OOP project, top-down design means starting with a problem description and working toward a solution using these guidelines:

1. Decide on the classes needed. You should normally include a driver class as one of the classes. To determine the other classes, think of the problem in terms of its component objects. Specify one class for each unique type of object. With large systems that have many classes, pure top-down design defers identification of detailed classes until later because identifying detail classes is itself a detail.
2. For each class, decide on its instance variables, which should be state variables identifying object attributes. The driver class should not have any instance

variables.

3. For each class, decide on its public methods. The driver class should contain only one public method—main.

4. For each public method, implement in a top-down fashion. Consider each public method to be a “top” method. If it is fairly involved and can be broken into subtasks, have it call private helper methods to do the subtask work. Finish writing the top methods before starting to write the lower level helper methods. Initially, implement the helper methods as *stubs*. A stub is a dummy method that acts as a placeholder for an actual method. A stub’s body typically consists of a print statement that displays something like “In method x, parameters = a, b, c” where x is the name of the method and a, b, and c are values of passed-in arguments. If it’s not a void method, the stub will also include a default return statement. You’ll see examples of void-method stubs later in this section.

Start testing right away.

5. Test and debug the program. The suggested stub print messages will help you trace the program’s actions.

6. Replace stub methods one at a time with fully implemented helper methods. After each replacement, test and debug the program again.

Top-down design is sometimes referred to as *stepwise refinement*. The term stepwise refinement is used because the methodology encourages programmers to implement solutions in an iterative manner where each solution “step” is a refined version of a previous solution step. After implementing top-level tasks, the programmer goes back and refines the solution by implementing the subtasks at the next lower levels.

Benefits of Using Top-Down Design

In top-down design, the designer doesn't worry initially about the details of subtask implementation. The designer focuses on the "big picture" first. Because it focuses on the big picture first, top-down design is good at getting a project going in the right direction. That helps to ensure that the completed program matches the original specifications.

Top-down design is particularly appropriate when a project involves many programmers. Its early emphasis on the big picture forces the project's programmers to agree on common goals. Its strong organizational emphasis promotes coherence and prevents the project from splintering off in different directions. The top-down design methodology facilitates tight managerial control.

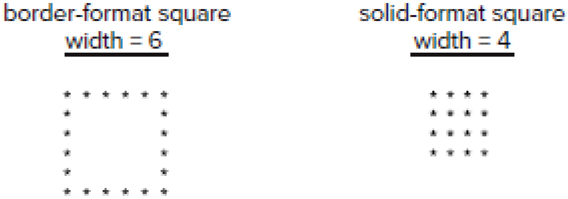
Square Program Example: First-Cut Version

Let’s now apply the top-down design methodology to a simple example. We’ll implement a Square class such that each Square object can:

Devise a way to solve the problem.

- Initialize the square's width.
- Calculate and return its area.
- Draw itself with asterisks using either an asterisks border or a solid pattern of asterisks. Each time the square is drawn, the user is prompted as to whether he or she would like a border format or

a solid format, like one of these:



Using the above top-down design guidelines, the first step is to decide on the classes. In this simple example, it’s easy to identify all the classes right at the start—SquareDriver and Square. The next step is to decide on the instance variables. They should be a minimum definitive set of object properties—state variables. All you need to specify a square is one number. The typical number that people use is the width. So we’ll use width as our lone instance variable.

⚠ But what about the square’s area? Area is a property, but it’s a simple function of width: area equals width squared. Because we can easily calculate area from width, it would be redundant to include area as another state variable. In principle, we could use area as the only state variable, and calculate width as the square root of area any time we needed width. But computing the square root is more difficult than computing the square, and we would frequently end up with a noninteger value for width, which would be hard to display in our prescribed asterisk format. So, for our problem, it's a better strategy to use width as the lone instance variable.

Identify state variables.

What about the solidness of the square? This is a conceptual choice. If you want to think of solidness as an inherent property of Square-class objects, it's appropriate to create another instance variable like boolean solid. On the other hand, if you want to think of solidness as just a temporary display option, solidness should not have state-variable status and it should not be an instance variable. For our example, we've elected to think of solidness as just a temporary display option, so we do not include it as another instance variable.

Returning to the top-down design guidelines, we see that the next step is to decide on the public methods. The problem description often determines what needs to be public. Here's what we need:

- a constructor that sets the square's width
- getArea—compute the square's area
- draw—display the square with asterisks using either an asterisks border or a solid pattern of asterisks

Let's now step back and look at what we've done so far. See Figure 8.9. It presents a first-cut UML class diagram for our solution's classes, instance variables, and constructor and public methods.

The next step in the top-down design process is to implement the main method in the top-level class This implementation appears in Figure 8.10. The code in main includes calls to the Square constructor and methods identified in Figure 8.9, but it does not yet say anything about how those members of the Square class are implemented.

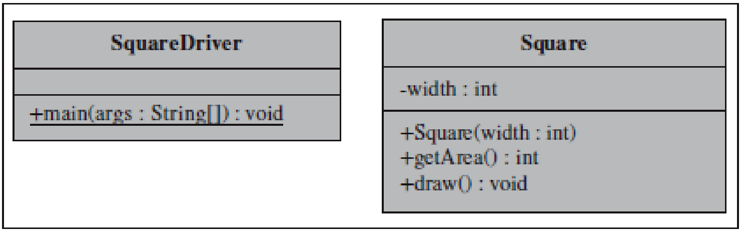
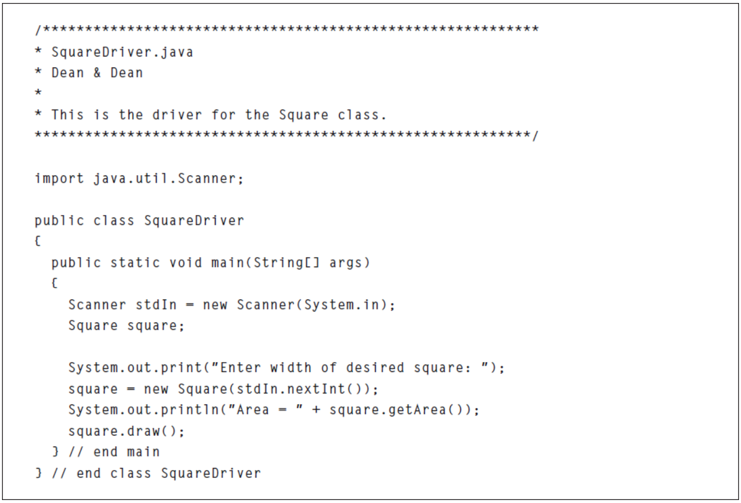

SquareDriver
+main(args : String[]) : void
Square
-width : int
+Square(width : int)
+getArea() : int
+draw() : void

Figure 8.9 Square program's UML class diagrams: first-cut version

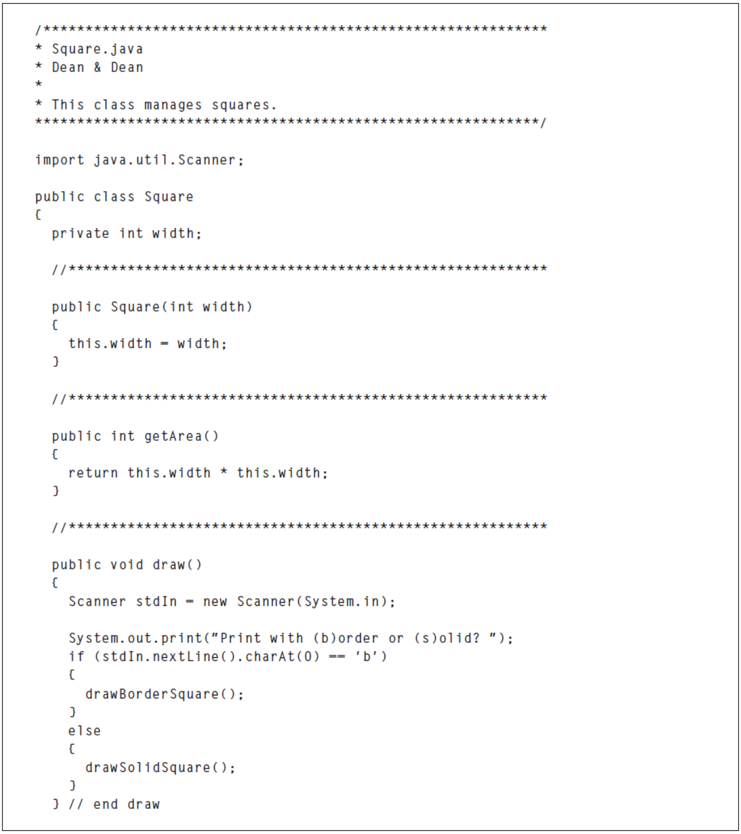


```
/*************************************************************
* SquareDriver.java
* Dean & Dean
*
* This is the driver for the Square class.
*************************************************************/

import java.util.Scanner;

public class SquareDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Square square;

    System.out.print("Enter width of desired square: ");
    square = new Square(stdIn.nextInt());
    System.out.println("Area = " + square.getArea());
    square.draw();
  } // end main
} // end class SquareDriver
```

Figure 8.10 SquareDriver class

The next step is to implement the public methods in the Square class. This implementation appears in Figure 8.11a. The constructor and getArea methods are straightforward and do not need explanation. But notice that the get in getArea makes this method look like an accessor that simply retrieves an instance variable. Is it OK to create this "false" impression? Yes, it is, because the instance variable is private and therefore hidden from public view. In fact, as noted above, we might actually have used area as the lone instance variable! A user of a class does not have to know exactly how it's implemented. Don't worry about the implementation when you pick a method name. It's the effect that matters, and getArea accurately describes the effect of calling that method.

The draw method prompts the user to choose either a border format or a solid format for the square's display. It's now becoming apparent that the draw method is not trivial. The drawBorderSquare and drawSolidSquare method calls are examples of subtasks that we should split off into separate helper methods.



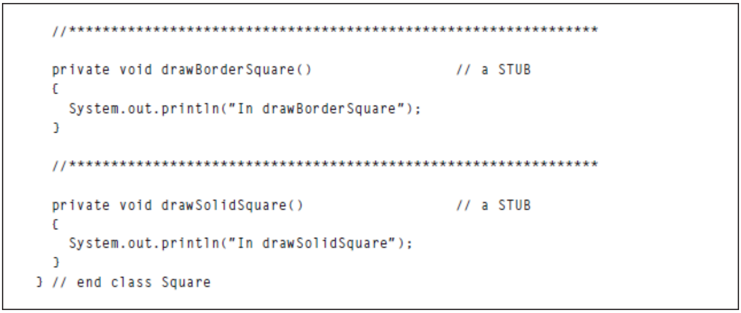
```
/*************************************************************
* Square.java
* Dean & Dean
*
* This class manages squares.
*************************************************************/

import java.util.Scanner;

public class Square
{
  private int width;

  //**********************************************************

  public Square(int width)
  {
    this.width = width;
  }

  //**********************************************************

  public int getArea()
  {
    return this.width * this.width;
  }

  //**********************************************************

  public void draw()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print(”Print with (b)order or (s)olid? ”);
    if (stdIn.nextLine().charAt(0) == 'b')
    {
      drawBorderSquare();
    }
    else
    {
      drawSolidSquare();
    }
  } // end draw
```

Figure 8.11a Square class: first-cut version—part A

Stubs

Top-down design tells us to implement helper methods initially as stubs. For our Square program, that means implementing drawBorderSquare and drawSolidSquare as stubs. Note the stubs in Figure 8.11b.

As you can probably surmise from the examples, a stub doesn't do much. Its main purpose is to satisfy the compiler so that the program is able to compile and run. Its secondary purpose is to provide an output that confirms that the method was called, and (where appropriate) show values passed into that method. When the stubbed Square program runs, it produces either this sample session:



```
  //**************************************************************

  private void drawBorderSquare()                 // a STUB
  {
    System.out.println("In drawBorderSquare");
  }

  //**************************************************************

  private void drawSolidSquare()                  // a STUB
  {
    System.out.println("In drawSolidSquare");
  }
} // end class Square
```

Figure 8.11b Square class: first-cut version—part B

```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? b
In drawBorderSquare
```

or this sample session:

```
Enter width of desired square: 5

Area = 25.0

Print with (b)order or (s)olid? s

In drawSolidSquare
```

Test one thing at a time.

Using stubs lets programmers test their partially implemented programs to determine whether their behavior is correct down to the stub level. Second, it makes debugging easier. After compiling and running the program successfully with stubs, replace the stubs with actual code one method at a time. As each stub is replaced, test and debug the updated program. If a bug appears, it should be easy to find because you know it’s probably in the most recently replaced method.

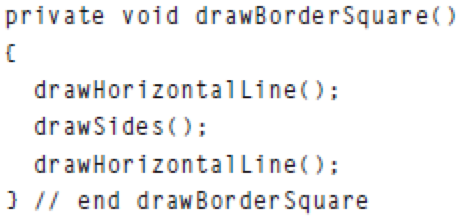
Square Program Example: Second-Cut Version

The next step in the top-down design process is to replace the helper methods' stub implementations with actual implementations. We have two helper methods to work on—drawBorderSquare and drawSolidSquare.

Let's start with the drawBorderSquare helper method. It prints a horizontal line of asterisks, prints the square's sides, and then prints another horizontal line of asterisks. Here's pseudocode for this algorithm:

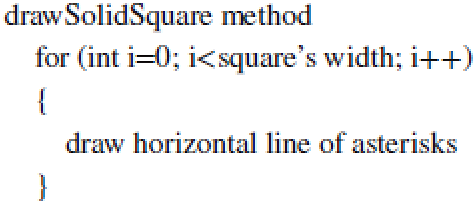
```
drawBorderSquare method
    draw horizontal line of asterisks
    draw sides
    draw horizontal line of asterisks
```

All three of drawBorderSquare's draw statements represent nontrivial tasks. Thus, when we translate the drawBorderSquare pseudocode into a Java method, we use method calls for each of the draw subtasks:



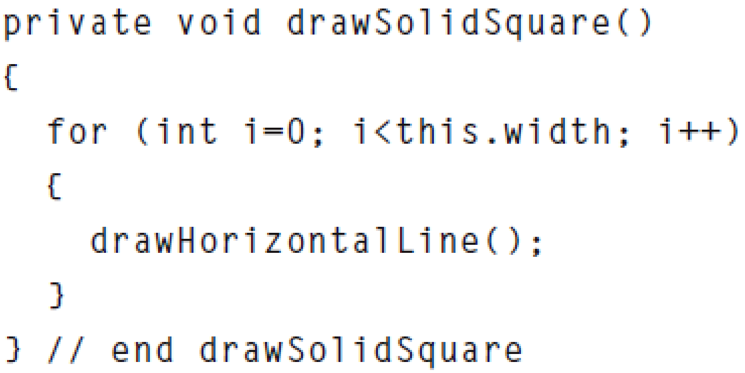
```
private void drawBorderSquare()
{
  drawHorizontalLine();
  drawSides();
  drawHorizontalLine();
} // end drawBorderSquare
```

Now let’s consider the drawSolidSquare helper method. It prints a series of horizontal lines of asterisks. Here’s pseudocode for its algorithm:



```
drawSolidSquare method
  for (int i=0; i<square’s width; i++)
  {
    draw horizontal line of asterisks
  }
```

Again, the draw statement represents a nontrivial task. Thus, when we translate the drawSolidSquare pseudocode into a Java method, we use a repeated method call for the draw subtask:

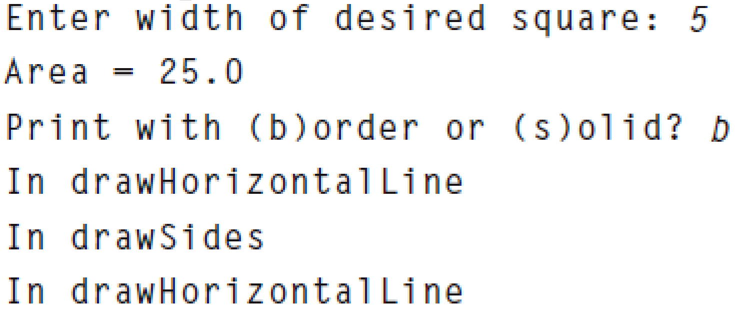


```
private void drawSolidSquare()
{
  for (int i=0; i<this.width; i++)
  {
    drawHorizontalLine();
  }
} // end drawSolidSquare
```

Notice that the drawBorderSquare method and the drawSolidSquare method both call the same drawHorizontalLine helper method. Being able to share the drawHorizontalLine method is a nice reward for our diligent use of helper methods, and it provides a good example for this general principle:

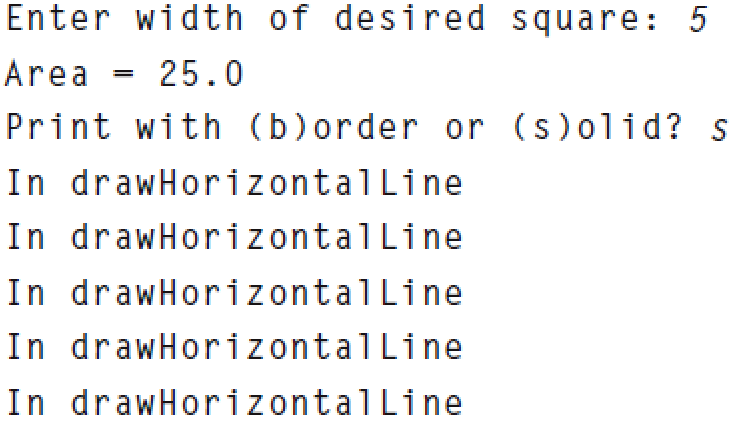
> If two or more methods perform the same subtask, avoid redundant code by having those methods call a shared helper method that performs the subtask.

By writing final code for the drawBorderSquare and drawSolidSquare methods and writing stub code for the drawHorizontalLine and drawSides methods, we complete the coding for the Square program's second-cut version. When executed with appropriate print statements in the two stub methods, drawHorizontalLine and drawSides, the second-cut version produces either this sample session:



```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? b
In drawHorizontalLine
In drawSides
In drawHorizontalLine
```

or this sample session:



```
Enter width of desired square: 5
Area = 25.0
Print with (b)order or (s)olid? s
In drawHorizontalLine
In drawHorizontalLine
In drawHorizontalLine
In drawHorizontalLine
In drawHorizontalLine
```

Square Program Example: Final Version

Keep documentation current.

To facilitate management, it’s a good idea to formalize your program’s design at various points during the design process. The formalization usually takes the form of UML class diagrams. Having up-to-date UML class diagrams helps to ensure project coherence. At a minimum, current UML class diagrams ensure that all members of a project are using the same classes, instance variables, and method headings. See Figure 8.12.

Figure 8.12 presents a UML class diagram for our complete Square program. It’s the same as our earlier UML class diagram except that we’ve added the helper methods.

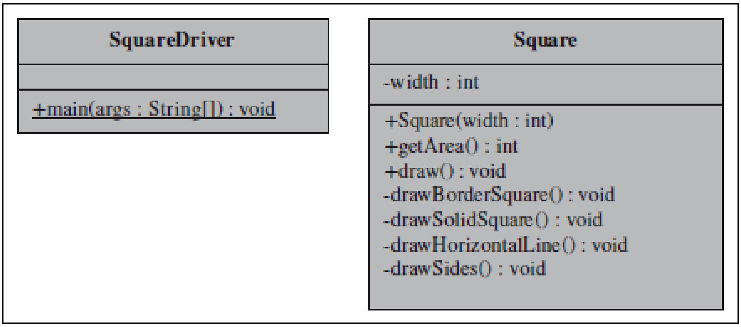
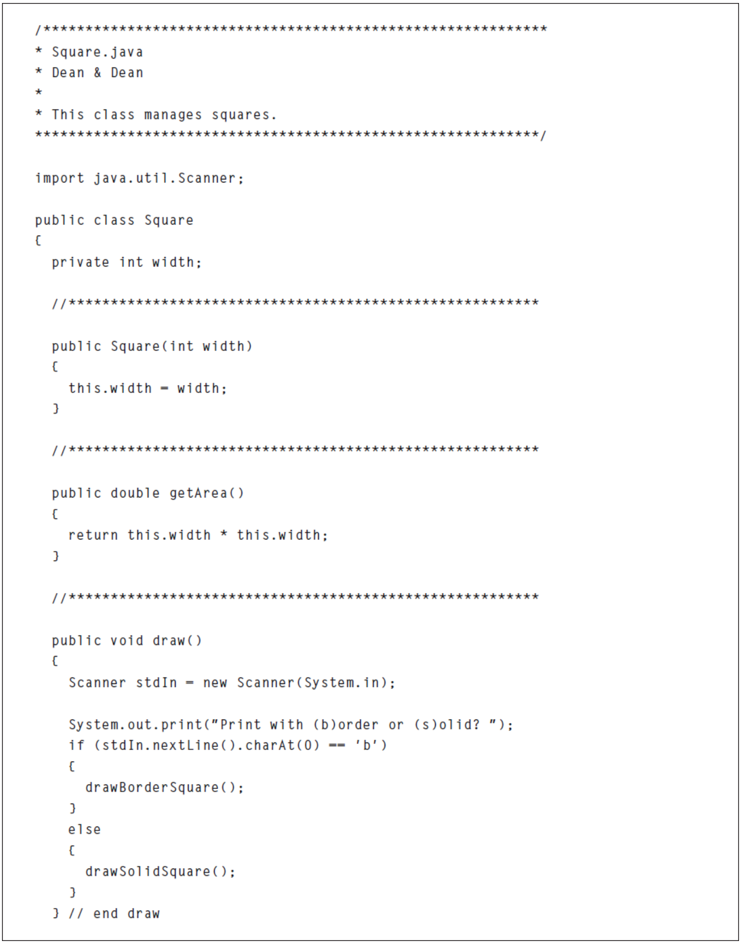

SquareDriver
+main(args : String[]) : void
Square
-width : int
+Square(width : int)
+getArea() : int
+draw() : void
-drawBorderSquare() : void
-drawSolidSquare() : void
-drawHorizontalLine() : void
-drawSides() : void

Figure 8.12 Square program's UML class diagram: final version

The second-cut version of the Square program contains stub implementations for the drawHorizontalLine and drawSides methods. Now, we need to replace those stub methods with actual methods. Figures 8.13a and 8.13b contain our final-version Square class. The only new items are the drawHorizontalLine and drawSides methods, which are straightforward. We encourage you to study their implementations on your own in Figure 8.13b.

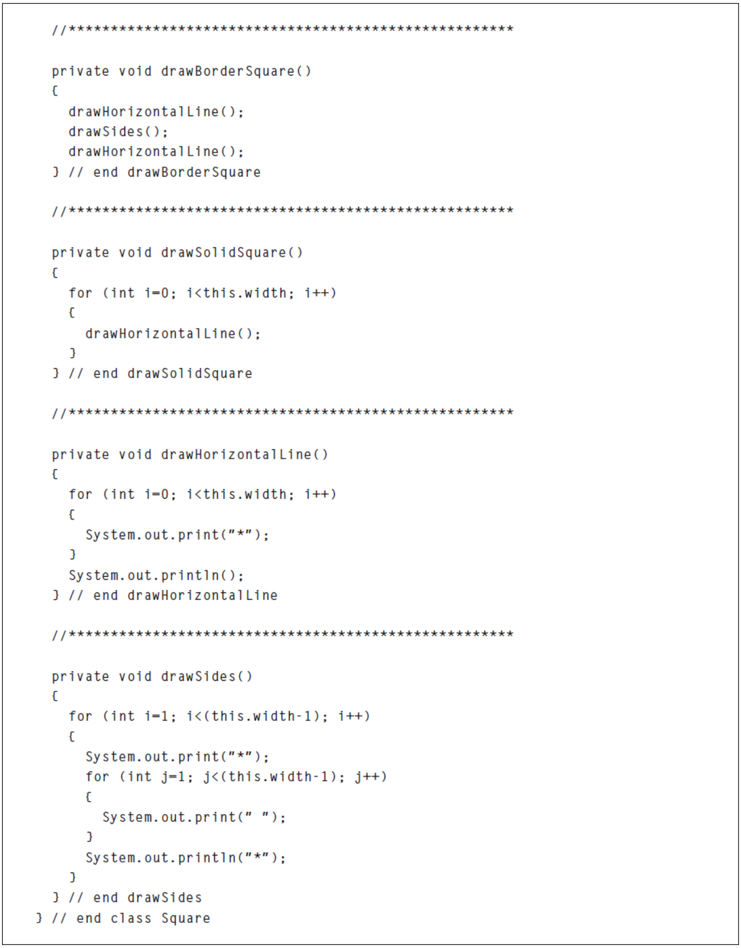
Top-Down-Design Downside

Almost every human-designed project must necessarily include some form of top-down thinking. However, pure top-down design has some undesirable side effects. One of these is that subordinate modules tend to be overly specialized. A well-known and particularly egregious example of how the top-down way of thinking can lead to excessive specialization is the case of the $660 Pentagon ashtrays. The Pentagon (headquarters of the U.S. Department of Defense) hired a large military contractor to manufacture ashtrays for Pentagon use. Because compatibility is important for many military components, the military generally wants faithful adherence to its specifications, and contractors naturally develop procedures and attitudes that promote conformity. However, sometimes there can be too much of a good thing. The ashtrays conformed perfectly to their specification, but each one had a price of $660. Top-down design went to a ridiculous extreme. Even though some of the top-level specifications may have been unconventional, the contractor probably followed the standard operating procedure and tried to match them perfectly. Hypothetical quote from the contractor's marketing manager: "What was specified did not match anything that was available, so we had to make it by hand in the machine shop."



```
/*************************************************************
* Square.java
* Dean & Dean
*
* This class manages squares.
*************************************************************/

import java.util.Scanner;

public class Square
{
  private int width;

  //**********************************************************

  public Square(int width)
  {
    this.width = width;
  }

  //**********************************************************

  public double getArea()
  {
    return this.width * this.width;
  }

  //**********************************************************

  public void draw()
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print(”Print with (b)order or (s)olid? ”);
    if (stdIn.nextLine().charAt(0) == ’b’)
    {
      drawBorderSquare();
    }
    else
    {
      drawSolidSquare();
    }
  } // end draw
```

Figure 8.13a Square class: final version—part A (an exact copy of Figure 8.11a)



```
  //*****************************************************

  private void drawBorderSquare()
  {
    drawHorizontalLine();
    drawSides();
    drawHorizontalLine();
  } // end drawBorderSquare

  //*****************************************************

  private void drawSolidSquare()
  {
    for (int i=0; i<this.width; i++)
    {
      drawHorizontalLine();
    }
  } // end drawSolidSquare

  //*****************************************************

  private void drawHorizontalLine()
  {
    for (int i=0; i<this.width; i++)
    {
      System.out.print("*");
    }
    System.out.println();
  } // end drawHorizontalLine

  //*****************************************************

  private void drawSides()
  {
    for (int i=1; i<(this.width-1); i++)
    {
      System.out.print("*");
      for (int j=1; j<(this.width-1); j++)
      {
        System.out.print(" ");
      }
      System.out.println("*");
    }
  } // end drawSides
} // end class Square
```

Figure 8.13b Square class: final version—part B (a fleshed-out version of Figure 8.11b)

You may be thinking—Interesting story, but how do the $660 ashtrays relate to programming? The top-down philosophy can lead to inefficient development practices. In the extreme case, that philosophy led to the military contractor expending enormous effort on the design and manufacture of something as simple as an ashtray. In general, the top-down design philosophy can motivate people to “reinvent the wheel.” This tends to increase overall product cost. It also tends to reduce the reliability of the final product. Why? Because with everything being new or reinvented, there’s no past history of testing and debugging to rely on. ⚠

8.9 Bottom-Up Design

Now, let’s look at the logical opposite of top-down design—bottom-up design. Bottom-up design implements specific low-level tasks first. To apply bottom-up design to the Square program, you might implement a drawSolidSquare method first. Next, you might implement a drawBorderSquare method. After finishing these bottom-level methods, you would implement higher-level methods, which are in charge of more general tasks, like a draw method to draw either type of square—a solid square or a border square.

As you implement each program component, you should test it immediately with a custom driver that's tailored to that particular component. You won't need any stubs, since already-tested lower-level methods will be available to be called by whatever higher-level method you are currently testing.

For simple programs like many of those that appear throughout the body of this book, bottom-up design is an appropriate strategy to use because it allows you to focus quickly on the essence of whatever problem is currently most critical, and it allows you to defer presentation details until later. For an example of bottom-up design, look at any program in this book in which we present a driven class before we present a driver for that class. Whenever we do that, we are using a bottom-up presentation, and we are inviting you to think about the program being described from the bottom up.

Bottom-up design also makes it easiest for you to use prewritten software, like that in the Java API and described previously in Chapter 5. The Java API is a particularly good source for prewritten software because its code is (1) optimized for high speed and low memory consumption and (2) highly reliable because it has undergone testing and debugging for years. It's good to use the Java API, but it takes time to learn how to use it. To learn about the Java API, see, for example, https://docs.oracle.com/en/

java/javase/13/docs/api/index-files/index-1.html. There, you'll find several ways to look things up. Here are two techniques:

1. Click "All Classes" and try guessing the name of a class that seems appropriate. Use the scrollbar in the classes frame to search for the guessed class name. There are about 4,500 classes, so finding a particular class requires a well-behaved mouse (we recommend proper diet and exercise to keep your mouse running smoothly). When you find a class name that looks promising, click on it and read about its public constants and methods.
2. Related classes are grouped together in about 230 packages. Click "All Packages" and use the scrollbar in the packages frame to find a package that looks

promising. Click on that package and scroll through its classes. Again, when you find a class name that looks promising, click on it and read about its public constants and methods.

Using prewritten software for your low-level modules reduces development time and project cost. It also improves product quality, because presumably the prewritten parts of your program have already been thoroughly tested and debugged. As in the case of Java API code, you'll often find that prewritten low-level software is quite flexible because it was designed for a broad spectrum of applications. This inherent low-level flexibility will make it easier for you to expand the capabilities of your program when you upgrade it in the future. Using prewritten software can facilitate parallel development. If several different programmers want to use a common subordinate module, they can do it independently. They do not have to coordinate their efforts because that module's design is already established and stable.

Work on the most critical problem first.

Another benefit of bottom-up design is that it provides the freedom to implement tasks in the most beneficial order. If there's a significant concern as to whether a particular calculation is feasible, it's important to begin working on that calculation as soon as possible. With bottom-up design, there's no need to wait around to address the concern—just attack it immediately. That way, you can determine at the earliest possible time whether the concern will be a showstopper. Likewise, if there is some low-level task that will take a long time to complete, bottom-up design allows you to begin work on it immediately and avoid a potential bottleneck later.

⚠ There are several drawbacks to using bottom-up design, however. As compared to top-down design, bottom-up design provides less structure and guidance. It’s often hard to know where to start, and because development is hard to predict, large bottom-up programming projects are hard to manage. In particular, with less inherent guidance, it’s harder for managers to keep their programmers on track. As a result, programmers might spend significant amounts of time working on code that may not be relevant to the final program. Another drawback in using bottom-up design is that it can lead to difficulties in getting the final product to conform precisely to design specifications. Top-down design facilitates conformity by addressing specifications in detail at the beginning. With bottom-up design, specifications receive only superficial consideration at the beginning.

So, when should you use bottom-up design? When you can use a substantial amount of prewritten and pretested low-level software, the bottom-up design process makes it easy for you to design around that software so that it fits naturally into your complete program. When you can use a substantial amount of prewritten software that is open to your inspection and already designed to fit together (like Java API software[7]), bottom-up design simultaneously promotes high quality and low cost. When low-level details are critical, bottom-up design motivates you to deal with the tough problems first—it gives you the most time to solve them. Thus, bottom-up design can also help you minimize delivery time.

A familiar example of bottom-up

software design is the early development of the Microsoft Windows operating system. The original version of Windows was built on top of the already existing and successful DOS operating system.8 The next major version of Windows was built on top of a novel low-level software core called “NT” (for New Technology). It’s important to note that the component source code in these cases was always open to and under the control of system developers, because it was all owned by the same company.9

8.10 Case-Based Design

There is another basic way to solve problems and design things. It’s what normal people do most of the time in their everyday lives. Instead of going through a formal top-down or bottom-up sequence of steps, you look around for an already-solved problem that's like the problem at hand. Then you figure out how that problem was solved, and you modify that solution to fit your problem. This approach is holistic. It starts with a whole solution and "bends" that whole solution to a different application.

If you have access to source code and the right to copy it or modify it and then redistribute it in a new context, you can modify an existing program or significant parts of existing code. Sometimes the code that you want to borrow is code you wrote yourself for a different application. Such code deserves your consideration, because you'll be intimately familiar with what it does and how it does it. For example, many of this book's projects were designed to show you how to solve a wide range of real-world problems. You can use the algorithms presented in the project assignments to generate Java code that solves particular versions of those problems. Once you have written that code, you'll be completely free to modify it and reuse it in any other context to solve other variations of those problems.

Frequently, the code that you'd like to use will be code that somebody else wrote. Would it be theft or plagiarism to use such code? It might be. If the code is copyrighted, and you don't have permission to use it, you shouldn't try to use it. But you might have permission to use it. Whenever you use code that somebody else wrote, be sure to acknowledge and identify your source.

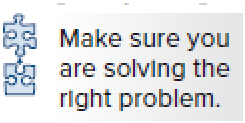
There is a growing body of what's called "free" software[10] that is debugged and maintained by a select body of experts, and it's available to all people to use and modify for their own purposes, provided they conform to certain reasonable rules: You may charge for products and services that use such software, but you must acknowledge its source, and you may not inhibit others from freely using it. Sometimes this software is low-level code that you can use like Java API software. But sometimes it's a complete program, which you can adapt to a problem you're currently addressing.

8.11 Iterative Enhancement

Often, you have to start working on a problem in order to understand how to solve the problem. That leads to a design process that is often iterative in nature. In the first iteration, you implement a bare-bones solution to the problem. In the next iteration, you add features and implement an improved solution. You continue adding features and repeating the design process until you have implemented a solution that does everything you need. This repetitive process is called *iterative enhancement.*

Prototyping—An Optional First Step

A *prototype* is a very “thin” or “bare-bones” implementation or perhaps just a faked “simulation” of a prospective program. Because of a prototype’s limited scope, developers can produce prototypes relatively quickly and present them to customers very early in the development process.


Make sure you are solving the right problem.

A prototype helps users get an early feel for what it will be like to use the - -program—well before the program is finished. It helps clients provide early feedback that improves the quality of product specification. Thus, prototyping provides a valuable adjunct to the first part of the top-down design process, and it complements early work in a bottom-up design process. Without a prototype, there's always a risk that you'll solve the wrong problem. Even if you solve the problem with great elegance, if it's the wrong problem, the whole effort is a waste of time.

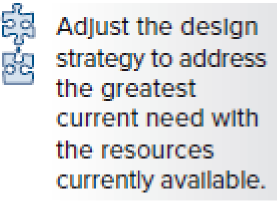
There are two basic ways to generate a prototype. One way is to write a very limited version of the final program in Java. Because a prototype should be relatively simple, you could use whatever design approach seemed easiest. The other way is to use a computer application that provides nice presentations to simulate the final program's user interface for particular "canned" data or a narrow range of user inputs.

⚠ Prototyping can be a valuable

communication tool, but use it with caution. Suppose that you create a prototype, show it to the client, and the client says: “I like it. Give me a copy so I can start using it tomorrow!” Don’t do it! If your prototype is an early iteration of an orderly sequence of planned iterations, fold in what you learn from client reaction, and proceed to the next iteration as originally planned. If your prototype is just a visual presentation pasted together from disparate components, resist the temptation to expand that prototype into a finished product. That’s tempting because you might think it would reduce development time. However, adding patches to a cobbled-together mock-up typically produces a messy result that is hard to maintain and upgrade. Eventually, it becomes necessary to rewrite massive amounts of code, and the associated confusion can destroy the program. It’s better to think of this kind of prototype as no more than a communication aid that elicits feedback that improves product specification.

Iterating

The first normal design iteration—or the iteration after an optional prototype—should be either a simple adaptation of some already-existing program or a bare-bones implementation developed with either the top-down or bottom-up design strategy. Subsequent iterations may or may not continue to use the same design strategy.


Adjust the design strategy to address the greatest current need with the resources currently available.

How do you decide which strategy to use for each iteration? Select that strategy which best addresses your greatest current need or concern:

- If your greatest current need is to understand what the customer wants, construct a prototype.
- If your greatest concern is on-time delivery, try to use an adaptation of existing software.
- If your greatest current concern is whether some particular functionality can be implemented, use the bottom-up design strategy to implement that functionality as soon as possible.
- If your greatest needs are reliability and low cost, use prewritten software with bottom-up design.
- If your greatest concern is overall performance and managerial control, use the top-down design strategy.

A famous iterated-design example is NASA's man-on-the-moon space program. President John F. Kennedy was thinking top-down when he announced the program. However, the first implementation was a prototype. Using a modified version of the existing Atlas ICBM rocket, “Project Mercury” shot one man a few hundred miles out into the Atlantic Ocean.

Subsequent iterations of Project Mercury used a bottom-up approach to put astronauts into Earth’s orbit. Then, NASA replaced the Atlas booster rocket with the newer and larger Titan ICBM rocket, which carried several people into Earth’s orbit in several iterations of “Project Gemini.”

NASA’s next iteration was a top-down design plan known as “Project Apollo.” Project Apollo originally envisioned the use of a gigantic booster rocket called Nova. After working on that for a while, NASA realized that a much smaller booster rocket (called Saturn) would suffice if a smaller moon lander was separated from the mother ship orbiting the moon, and the moon lander's return module was separated from its descent mechanism.

Project Apollo was a top-down design, optimized for NASA's requirements, rather than a bottom-up adaptation of existing military equipment. In the end, the top-down plan involving Nova was scrapped and replaced by a radically different top-down plan. This apparently erratic development sequence is a great example of successful real-world design. The history of successful software is the same. Different design cycles often emphasize different design strategies, and sometimes there are major changes.

Maintenance

After a program has been developed and put into operation, you might think there's no more need to work on it. Not so. In the real world, if a program is useful, programmers are often asked to *maintain* it long after that program is first put into operation. On average, 80% of the work on a successful program is done after the program is first put into operation. Maintenance consists of fixing bugs and making improvements. Maintenance is much easier if good software practices are employed at the beginning and throughout the life of the program. This includes writing the code elegantly in the first place, preserving elegance when you make changes, and providing and keeping complete and well-organized documentation.

Remember that documentation is more than just comments for programmers reading source code. Documentation is also interface information for programmers who want to use already-compiled classes.

Section 8.3 in this chapter and Appendix 6 show how to embed interface information in your source code so it can be read by javadoc and presented like Oracle's documentation of the Java API. Documentation also includes information for people who are not programmers at all but need to use a finished program. This type of documentation needs to be even more user-oriented than javadoc's output.

If you are responsible for maintaining an existing program, here are some useful thumb rules:

1. Respect your predecessor. Don't change any piece of code you think is wrong until you have spent as much time thinking about it as some other programmer (or you) spent creating it in the first place. There may have been an important reason for doing something in a certain way, even if there is a problem in how it was done, and you want to understand that reason before you make changes.

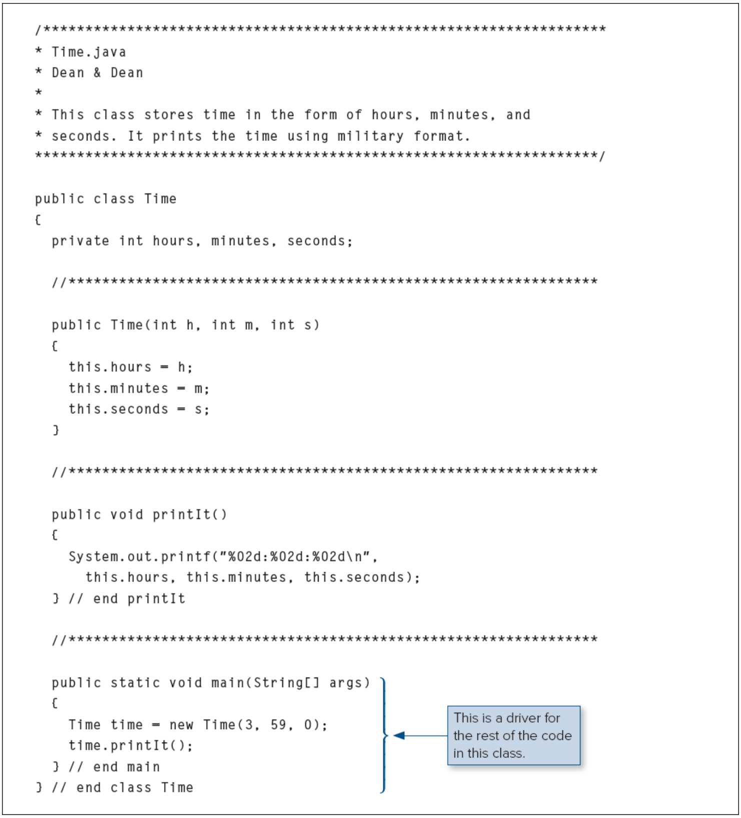
2. Respect your successor. Whenever you have trouble figuring out what a particular section of code is doing, after you thoroughly understand the problem, fix the code and documentation so that it is easier to figure out next time.

3. Maintain a “standard” bank of test input data (and the corresponding output data), and use it to verify that any changes you have made affect only the problem you are trying to solve and do not have other unwanted effects that ripple through the program.

8.12 Merging the Driver Method into the Driven

Class

It’s legal to include a main method in any class. Figure 8.14 contains a simple Time program that includes its own main method.



```
/*******************************************************************
* Time.java
* Dean & Dean
*
* This class stores time in the form of hours, minutes, and
* seconds. It prints the time using military format.
*******************************************************************/

public class Time
{
  private int hours, minutes, seconds;

  //***************************************************************

  public Time(int h, int m, int s)
  {
    this.hours = h;
    this.minutes = m;
    this.seconds = s;
  }

  //***************************************************************

  public void printIt()
  {
    System.out.printf("%02d:%02d:%02d\n",
      this.hours, this.minutes, this.seconds);
  } // end printIt

  //***************************************************************

  public static void main(String[] args)
  {
    Time time = new Time(3, 59, 0);
    time.printIt();
  } // end main
} // end class Time
```

This is a driver for the rest of the code in this class.

Figure 8.14 Time class with a built-in main driver method

Up until now, we’ve split each of our OOP programs into separate classes—a driver class and one or more driven classes. It’s easiest to grasp the concept of an object if it’s associated with one class, while the code that instantiates it is associated with another class. Driven classes and driver classes have distinctive roles. A driven class describes a thing that’s being modeled. For example, in our Mouse programs, the Mouse class describes a mouse. A driver class contains a main method, and it drives the separate Mouse class. In our Mouse programs, the MouseDriver class instantiates Mouse objects and performs actions on those objects. Using two or more classes fosters the habit of putting different types of things in different modules.

Although we’ll continue to use separate classes for most of our programs, for short programs that don’t do much except demonstrate a concept, we'll sometimes merge main into the class that implements the rest of the program. It's a matter of convenience—there's one less file to create and there's slightly less code to enter.

Provide each class with a built-in test method.

In a big program that has one driver class in charge of a large number of driven classes, it's sometimes handy to insert an additional main method in some or all of the driven classes. The additional main method in a driven class serves as a local tester for the code in that class. Whenever you make a change in the code of a particular class, you can use its local main method to test that class directly. It's easy. Just execute the class of interest, and the JVM automatically uses that class's main method. Once you've verified the changes you've made locally, you can proceed to execute the driver in a higher-level module to test more or all of the program. You don't have to remove the local main methods. You can just leave them there for future local testing or demonstration of the features of each particular class. When you execute the overall program's driver class, the JVM automatically uses the main method in that driver class, and it ignores any main methods that may happen to be in other classes in the program.

Thus, you can add a main method to any class, so the class can be executed directly and act as its own driver. When a multiclass program contains multiple main methods (no more than one per class), the particular main method that's used is the one in the class that's current when execution starts.

8.13 Accessing Instance

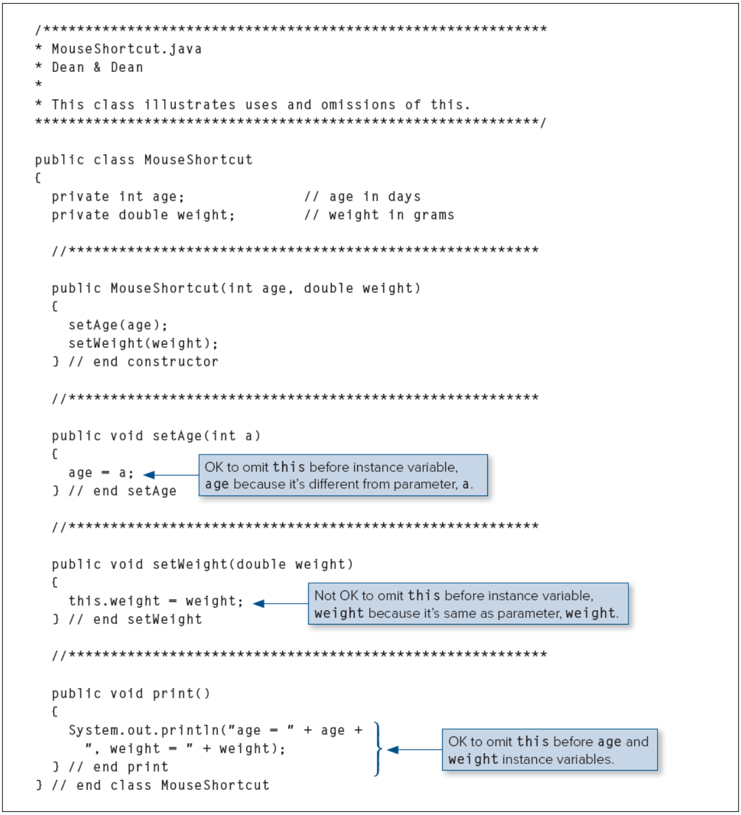
Variables Without Using this

For a while now, we've used this to access the calling object's instance variables from within a method. Here's a formal explanation for when to use this:

> Use this within an instance method or a constructor to access the calling object's instance variables. The this reference distinguishes instance variables from other variables (like local variables and parameters) that happen to have the same name.
>
> However, if there is no name ambiguity, you may omit the this prefix when accessing an instance variable.

The code in Figure 8.15 has several places where the this prefix is worth mentioning. It’s OK to omit this in the statement in the setAge method because the instance variable name is different from the parameter name. It’s not OK to omit this in the statement in the setWeight method because the identical instance variable and parameter names would create an ambiguity. It is OK to omit this in the statement in the print method because there is no name ambiguity.

Sometimes an instance method is called by one object and has a parameter that refers to a different object in the same class. String’s equals method is a familiar example of this situation. Inside such a method, there will be code that needs to refer to two different objects, the calling object and the object referred to by the parameter. The safest and most understandable way to refer to these two objects is to use the this prefix to refer to the calling object and the reference-parameter prefix to refer to the other object. However, it’s OK to omit the this when referring to the calling object, and you’ll see this done quite frequently. It makes the code more compact.



```
/*************************************************************
* MouseShortcut.java
* Dean & Dean
*
* This class illustrates uses and omissions of this.
*************************************************************/

public class MouseShortcut
{
  private int age;              // age in days
  private double weight;        // weight in grams

  //*********************************************************

  public MouseShortcut(int age, double weight)
  {
    setAge(age);
    setWeight(weight);
  } // end constructor

  //*********************************************************

  public void setAge(int a)
  {
    age = a;
  } // end setAge

  //*********************************************************

  public void setWeight(double weight)
  {
    this.weight = weight;
  } // end setWeight

  //**********************************************************

  public void print()
  {
    System.out.println("age = " + age +
      ", weight = " + weight);
  } // end print
} // end class MouseShortcut
```

OK to omit `this` before instance variable, `age` because it’s different from parameter, `a`.

Not OK to omit `this` before instance variable, `weight` because it’s same as parameter, `weight`.

OK to omit `this` before `age` and `weight` instance variables.

Figure 8.15 MouseShortcut class illustrates the use and omission of this

8.14 Writing Your Own Utility Class

Up to this point, you've implemented methods that solve problems for a particular class. Suppose you want to implement methods that are more general purpose, so that multiple and unforeseen classes can use them. Those types of methods are called *utility methods*. In the past, you've used utility methods from the Math class; for example, Math.round and Math.sqrt. You can also write your own utility methods and put them in your own utility class.

For example, Figure 8.16's PrintUtilities class provides print-oriented utility constants and methods. The two constants, MAX_COL and MAX_ROW, keep track of the maximum column and maximum row for a standard-sized piece of paper. If you have multiple classes that print reports, those constants can help to ensure report-size uniformity. The printCentered method prints a given string horizontally centered. The printUnderlined method prints a given string with dashes underneath it. These methods belong in a utility class because they perform print routines that might be needed by multiple other classes.

```
/*********************************************************************
* PrintUtilities.java
* Dean & Dean
*
* This class contains constants and methods for fancy printing.
*********************************************************************/

public class PrintUtilities
{
  public static final int MAX_COL = 80; // last allowed column
  public static final int MAX_ROW = 50; // last allowed row

  //*******************************************************************

  // Print given string horizontally centered.

  public static void printCentered(String s)
  {
    int startingCol; // starting point for string
    startingCol = (MAX_COL / 2) - (s.length() / 2);

    for (int i=0; i<startingCol; i++)
    {
      System.out.print(” ”);
    }
    System.out.println(s);
  } // end printCentered

  //*******************************************************************

  // Print given string with dashes underneath it.

  public static void printUnderlined(String s)
  {
    System.out.println(s);
    for (int i=0; i<s.length(); i++)
    {
      System.out.print(”-”);
    }
  } // end printUnderlined
} // end class PrintUtilities
```

Figure 8.16 Example utility class that handles special-needs printing

In the PrintUtilities class, note that the constants and methods all use the public and static modifiers. That’s normal for utility class members. The public and static modifiers make it easy for other classes to access PrintUtilities’ members.

8.15 Problem Solving with the API Calendar Class (Optional)

Don't reinvent the wheel.

Although textbooks (including ours) ask you to write little programs that manipulate times and dates, if you get serious about times and dates, you will discover it's a hornet's nest of different number bases, different length months, leap years, daylight savings time, different time zones, and many different formatting conventions. For serious time and date work, you should use Java API prewritten software. Unfortunately, it's not always easy to find the right Java class. For example, if you're working on a program that requires you to keep track of the date and time, you might be inclined to use the Date and Time classes, but those classes are obsolete. Usually, you should use the Calendar class instead. Figure 8.17 contains a CalendarDemo program that exercises some of the methods in the Calendar class.

The Calendar class is in the java.util package. To include it in your program, you could use this import statement:

```
import java.util.Calendar;
```

However, since the Calendar class is in the same package as the Scanner class, which this program also needs, it's easier to make both classes available simultaneously with this one "wildcard" import statement:

```
import java.util.*;
```

In the CalendarDemo program's time initialization statement, the time variable is assigned a reference to an instance of the Calendar class. Notice that instead of using new Calendar(), we get a Calendar instance by calling Calendar's getInstance method. If you look up the getInstance method in the Java API documentation for the Calendar class, you'll see that this method has a static modifier, so it's a static method. How do you invoke a static method? Think back to how you invoked methods from the Math class in Chapter 5. Instead of using an instance variable before the method name, you use the class name. How does getInstance work? You don't need to know, because it's an encapsulated module, but it's reasonable to assume it internally instantiates a Calendar object, initializes it with the current time, and then returns a reference to that object. Although this is not the standard way to instantiate new objects, it works. The Java API includes several examples of this indirect type of object construction.

For the rest of the program, you can forget about how the time object was created and use it as you would any other object to call instance methods in its own class. The first print statement uses Calendar's getTime method to retrieve the time information, and then it prints it all out as shown in the first line of the sample session.

Use ID number in argument to select one of many similar variables.

The next two statements use the object reference with get methods to retrieve two particular instance variable values. But wait! There's something wonderfully strange about these two get methods. They're not two separate methods like getDayOfYear and getHour would be. They're both the same method—one method called just plain get. Instead of using the method name to identify the instance variable that will be retrieved, the designers of this class decided to use an int parameter value to identify that variable. We don't have to know how the method is implemented, because it’s encapsulated, but we can use a plausible guess to shed light on what it does. For example, get’s parameter could be a switch index that steers the control flow to a particular case, where there’s code that returns the value of the instance variable that corresponds to that index number.

```
/*********************************************************************
* CalendarDemo.java
* Dean & Dean
*
* This program demonstrates how to use the Calendar class.
*********************************************************************/

import java.util.*;                 // for Scanner and Calendar

public class CalendarDemo
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Calendar time = Calendar.getInstance();    // initially now
    int day;                                   // day of year
    int hour;                                  // hour of day

    System.out.println(time.getTime());
    day = time.get(Calendar.DAY_OF_YEAR);  }
    hour = time.get(Calendar.HOUR_OF_DAY); }
    System.out.println("day of year= " + day);
    System.out.println("hour of day= " + hour);

    System.out.print("Enter number of days to add: ");
    day += stdIn.nextInt();
    System.out.print("Enter number of hours to add: ");
    hour += stdIn.nextInt();

    time.set(Calendar.DAY_OF_YEAR, day);
    time.set(Calendar.HOUR_OF_DAY, hour);
    System.out.println(time.getTime());
  } // end main
} // end class CalendarDemo
```

Parameters are `int` codes that specify the kind of information desired.

Sample session:

```
Sat Feb 15 13:20:40 CST 2020
day of year = 46
hour of day = 13
Enter number of days to add: 8
Enter number of hours to add: 13
Mon Feb 24 02:20:40 CST 2020
```

Figure 8.17 Demonstration program for the Calendar class

The problem with using an index number to identify one of many instance variables is that simple integers don't convey much meaning. But you know a solution to this problem. All you have to do is make each such index number a named constant. Then, for the distinguishing method argument, use the named constant instead of the number. That's how the Calendar class implements its generic get method. And it's at least as easy for a user to remember one get method with different named-constant arguments as it would be to remember different names for get methods.

Armed with this concept, you should now be able to see what the rest of the code in our CalendarDemo program is doing. It gets the current day of the year and the current hour of the day. Then it adds a user-input number of days to the current day and a user-input number of hours to the current hour. Then it mutates the object's instance variables for day-of-year and hour by using Calendar's generic set method. Finally, it prints out the mutated time.

The Calendar class nicely illustrates the value of using prewritten software. It really is easier to learn to use that class than it is to write a program that does what it does. Moreover, other people's code sometimes illustrates techniques that may be applicable to code that you write. However, the Calendar class also illustrates the kinds of penalties associated with using prewritten software. The biggest penalty is usually the time that you have to spend to locate and figure out what's available. Another penalty is that what you find may not exactly match your immediate needs, and you might have to provide extra code to adapt the prewritten software to your current program. Such penalties motivate many programmers to say, "Oh heck, I'll just write it myself." Sometimes that's the thing to do, but in the long run you'll be ahead if you take time to learn about what others have already developed.

8.16 GUI Track: Problem Solving with CRC Cards (Optional)

When you begin a new design, there's often a period of head-scratching when you're trying to figure out what your classes should be and what they should do. Section 8.8 presented a formal top-down recipe, but sometimes you just need to "muck around" or brainstorm for a while to get your thinking straight.

Even when you're just mucking around and brainstorming, it's still helpful to write things down. To provide a minimal structure for this informal activity, several years ago computer scientists Kent Beck and Ward Cunningham11 suggested using old-fashioned 3″ × 5″ index cards, with a pencil and eraser. Their idea was to allocate one card to each proposed class, with three kinds of information on each card: (1) At the top, put a class name. (2) Below and on the left, make a list of imperatives that say what you want that class to do. (3) Below and on the right, make a list of other classes with which the current class interacts—either actively as a client or passively as a server. The acronym, CRC, helps you remember the three kinds of information on each card:

C = Class
R = Responsibilities
C = Collaborators

In the old days, when several different people participated in a brainstorming session, pencils, erasers, and little white cards were indeed the best medium. But now, if you're the only designer, it might be more convenient to use little windows on your computer screen. The body of Chapter 16 and a GUI section at the end of Chapter 16 will show how a user can easily save (and later retrieve) tentative entries on such “cards” in computer files, which the user could exchange with coworkers through the Internet.

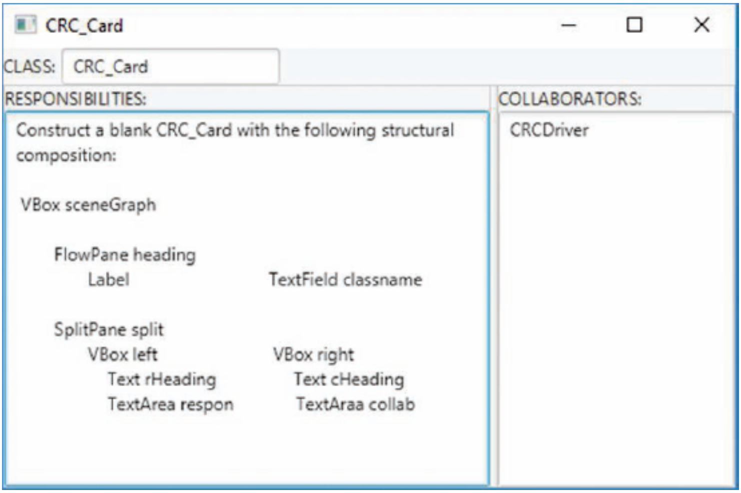
This section will get you started in designing such a system, using Java skills you have already plus some new GUI features. This section will utilize a top-down design process involving two classes—an initial driven class (the CRC card creator) and a primitive initial driver. The driven class uses some programming features presented in previous optional GUI sections, plus some new GUI features explained in this section. The driven class creates a GUI version of a single CRC card.

In a GUI section at the end of

Chapter 10, we reuse this section's driven class, and we revise this section's driver class. The revised driver will use both non-GUI techniques presented in the main body of Chapter 10 and GUI techniques presented in the GUI section at the end of Chapter 10. Then later, in the GUI section at the end of Chapter 16, we add get and set methods to this section's driven class and modify the driver again to enable transfer of user-entered data back and forth between the computer screen and files in the computer memory.

Figure 8.18 shows the card this section's driven class creates. The user is able to (1) modify the default class name in the text field to the right of the word "CLASS", (2) enter a list of responsibilities in the text area under the word RESPONSIBILITIES, and (3) enter a list of collaborators in the text area under the word

COLLABORATORS. In this case, under RESPONSIBILITIES, the user decided to do some self-documentation and make an outline of the structure the class's constructor is supposed to create.



CLASS: CRC_Card

RESPONSIBILITIES:

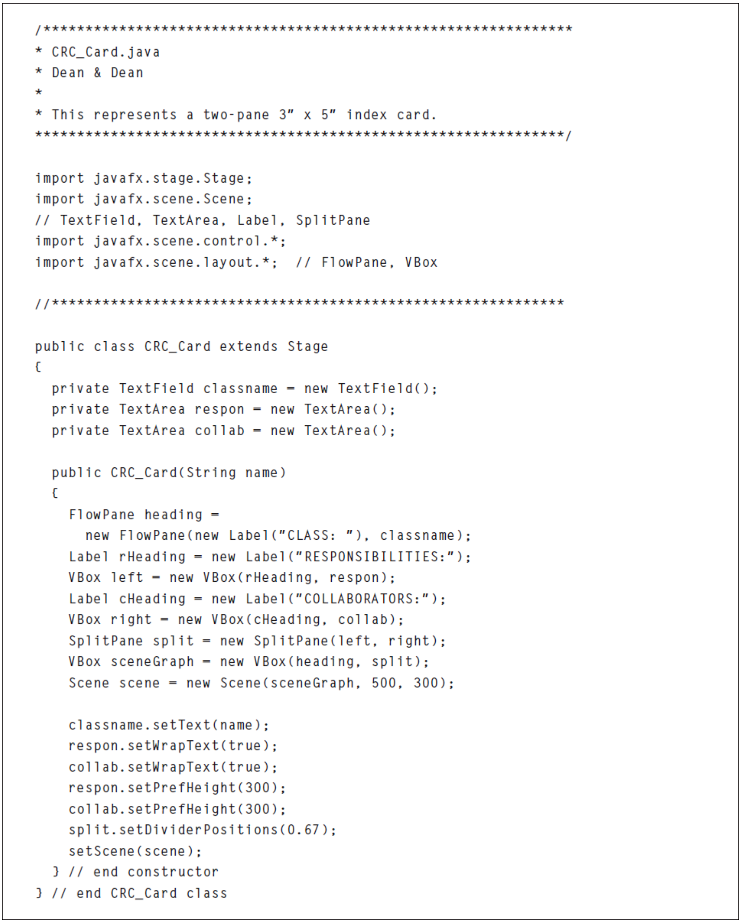
```
Construct a blank CRC_Card with the following structural
composition:

VBox sceneGraph

    FlowPane heading
        Label                   TextField classname

    SplitPane split
        VBox left               VBox right
            Text rHeading           Text cHeading
            TextArea respon         TextAraa collab
```

COLLABORATORS:

CRCDriver

Figure 8.18 Display produced by constructor of CRC Card class with subsequent user entries in the TextArea components under the RESPONSIBILITIES and COLLABORATORS headings. Figure 8.19 shows the code for CRC_Card, and Figure 8.20 shows the code for CRCDriver.

Figure 8.19 shows our initial iteration of the program's driven class, CRC_Card. It extends JavaFX's Stage class. In addition to its Stage import, it imports packages that provide access to several other prewritten JavaFX classes: Scene, TextField, TextArea, Label, SplitPane, FlowPane, and VBox. The GUI section at the end of Chapter 1 used the Label class, and the GUI sections of Chapter 5 and the Interlude used the Scene class. But this is the first time you will have seen TextField, TextArea, SplitPane, FlowPane, and VBox. These five classes are containers, and each of them has a special way of automatically organizing its contents.



```
/***************************************************************
* CRC_Card.java
* Dean & Dean
*
* This represents a two-pane 3” x 5” index card.
***************************************************************/

import javafx.stage.Stage;
import javafx.scene.Scene;
// TextField, TextArea, Label, SplitPane
import javafx.scene.control.*;
import javafx.scene.layout.*;  // FlowPane, VBox

//*************************************************************

public class CRC_Card extends Stage
{
  private TextField classname = new TextField();
  private TextArea respon = new TextArea();
  private TextArea collab = new TextArea();

  public CRC_Card(String name)
  {
    FlowPane heading =
      new FlowPane(new Label(”CLASS: ”), classname);
    Label rHeading = new Label(”RESPONSIBILITIES:”);
    VBox left = new VBox(rHeading, respon);
    Label cHeading = new Label(”COLLABORATORS:”);
    VBox right = new VBox(cHeading, collab);
    SplitPane split = new SplitPane(left, right);
    VBox sceneGraph = new VBox(heading, split);
    Scene scene = new Scene(sceneGraph, 500, 300);

    classname.setText(name);
    respon.setWrapText(true);
    collab.setWrapText(true);
    respon.setPrefHeight(300);
    collab.setPrefHeight(300);
    split.setDividerPositions(0.67);
    setScene(scene);
  } // end constructor
} // end CRC_Card class
```

Figure 8.19 CRC_Card class

This is the driven part of a program that displays a blank CRC card on the computer screen. It is driven by the CRCDriver class in Figure 8.20 and produces the card framework in Figure 8.18.

TextField and TextArea left-align their String contents. By default, if the String's length exceeds the container width, these containers shift their contents out of view to the left so the user can keep adding at the right end. When this happens in a TextArea, a horizontal scroll bar appears. With a TextArea, the programmer can optionally specify wrapping to the next line at ends of words. This keeps all previously written text in view until the text reaches the container’s bottom, at which point a vertical scroll bar appears.

A SplitPane allows the programmer to encapsulate multiple subordinate objects separated by user-movable dividers. By default, these objects are arranged horizontally, side by side. A FlowPane is like a wrap-enabled TextArea, except its contents may be almost any kinds of objects, not just the words in a String. Like a FlowPane, a VBox’s contents can be almost any kinds of objects, but it organizes its constructor's arguments vertically—from top to bottom—and there is no wrapping.

In Figure 8.19's CRC_Card code, we might have included the three instance variables (classname, respon, and collab) in the constructor as local variables. However, these variables are the central focus of this class. They are its *state* variables. Making them instance variables gives them the prominence they deserve. Besides, we expect a subsequent iteration might add get and set methods, which will require accessibility of these variables from outside the constructor. Therefore, it's doubly appropriate to make them instance variables.

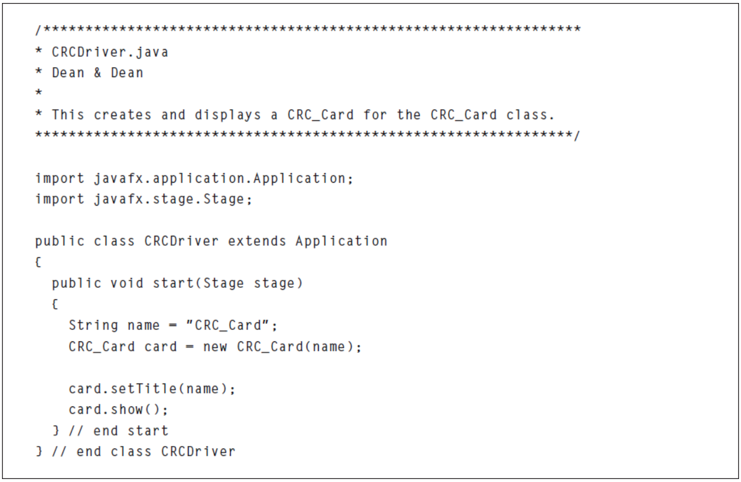
The CRC_Card constructor composes the card. At the top of each card is a horizontal container called FlowPane heading, which contains the "CLASS" text and the classname state variable. Below it is a SplitPane split container, which in turn contains subordinate VBox left and VBox right containers. The VBox left container contains the "RESPONSIBILITIES:" label and the respon state variable. The VBox right container contains the "COLLABORATORS:" label and the collab state variable. The FlowPane heading and SplitPane split containers are themselves components in a larger VBox called sceneGraph. A scene graph is the composition of everything in a stage's scene. The last declaration puts the scene graph into the scene and prescribes the scene's initial width and height in pixels.

The first statement after the declarations establishes a default value for the classname state variable, making it equal to the card's name. Although the user can change classname at any later time, this constructor enables a driver to automatically initialize it by supplying a string value in the instantiation process. The two setWrapText(true) statements enable text wrapping in the TextArea components under the “RESPONSIBILITIES:” and “COLLABORATORS:” headings. The next two statements set preferred heights for the two TextArea containers. The setDividerPositions statement makes the left text area twice as wide as the right text area initially, but the user can change their relative widths by using the mouse to drag the divider either left or right. The CRC_Card constructor’s final statement sets the scene on the CRC_Card Stage.

Figure 8.20 shows our initial iteration of the program’s driver. Like the GUI programs in Sections 1.10, 3.25, and 5.9 and in the Interlude’s second section, this driver imports the JavaFX Application and Stage classes, extends Application, and overrides

Application's start method.

All JavaFX programs must include a class that extends Application and must include an overriding start method with a Stage stage argument. Appication's hidden software creates this primary stage automatically and passes it to the program's start method. This automatically created stage object appears on the screen when there is an explicit stage.show method call, as in the TitleHello and LabelHello programs in Chapter 1's GUI section, the GraphicsDemoC program in Chapter 5's GUI section, and the GraphicsDemoJ program in the Interlude's GUI section. But that stage object does not appear on the screen in the DialogDemo program in Chapter 3's GUI section. In this latter case, what appears on the screen are special stages created in the background by prewritten JavaFX dialog classes, and the DialogDemo program made them appear with its explicit showAndWait method calls. This section's GUI program illustrates yet another alternative. Its CRC_Card class is itself a Stage class because it extends JavaFx's Stage class. Our current program's driver employs card.show() to display the driven CRC_Card's stage, but it does not show its own stage.



```
/*******************************************************************
* CRCDriver.java
* Dean & Dean
*
* This creates and displays a CRC_Card for the CRC_Card class.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;

public class CRCDriver extends Application
{
  public void start(Stage stage)
  {
    String name = "CRC_Card";
    CRC_Card card = new CRC_Card(name);

    card.setTitle(name);
    card.show();
  } // end start
} // end class CRCDriver
```

Figure 8.20 CRCDriver class

This is the driver of a program that displays a CRC card on the computer screen. It drives the CRC_Card class in Figure 8.19 to produce output like Figure 8.18, except that the output would have no user entries.

Summary

- Begin every class with a prologue. Include program name, author(s), and a brief description of what the class does.
- Provide a descriptive comment above or after any code that an experienced Java programmer would not understand.
- Use meaningful names for everything. Do not be cryptic.
- Enclose logical blocks of code in braces. The opening and closing braces should be in the same column as the start of the line that precedes the opening brace.
- Supply a // end <*block name*> comment after a block's closing brace to improve readability.
- Declare each variable at the beginning of its class or method,

or in its for loop header. Normally use one line per variable and follow each declaration with an appropriate descriptive comment.

- When a method won't work as might be expected unless certain conditions exist before it's called, warn users with a precondition comment just above the method header.

- When a method modifies variables in ways that might not be expected, warn users with a postcondition comment just above the method header.

- Use javadoc with */** comment */* above constructor and method headers to generate documentation like Java API documentation.

- Practice mentally putting

yourself in a user's shoes. Test prospective outputs and inputs with users, not only to tune the cosmetics but also to check basic assumptions and objectives.

- Early in the planning process, separate project concerns into categories like entity, relationship, structure, behavior, and appearance, and try to associate each concern with an appropriate feature of your programming language.

- Use subordinate helper methods to simplify large methods and reduce code redundancy. Make helper methods private to minimize clutter in the class interface.

- Use instance variables for object attributes (state information) only. Use local variables and input parameters for calculations within a method and to transfer data into a method. Use return values and/or input reference parameters to transfer data out of a method.

- Plan to test the software you develop frequently and thoroughly as you go along. Include typical, boundary, and unreasonable cases.
- Top-down design is appropriate for large projects that have well-understood objectives. Proceed from general to specific, using stubs to defer implementation of subordinate methods.
- Bottom-up design allows you to give priority to critical details. It fosters reuse of existing software, which reduces development costs and improves system reliability. But this methodology makes large projects hard to manage.
- Expect to go through several design iterations. Use prototyping

to help customers get a clearer understanding of what they want, but avoid the trap of trying to convert a clumsy prototype directly into a final product. In each subsequent iteration, select that design strategy which best addresses the greatest current need or concern. A successful program will require ongoing maintenance, and you can make this easier if you preserve and enhance elegance as the program changes and grows.

- To facilitate modular testing, provide a main method with every class.
- If there is no name ambiguity, you may omit the this prefix when accessing an instance member.
- When you need to perform the same calculations in different applications, put them in public

static methods in a custom utility class.

Review Questions

§8.2 Coding-Style Conventions

1. One should avoid inserting blank lines between different code sections (because that leads to wasted paper when the program is printed). (T / F)

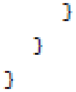
2. In order, list the seven items that we recommend you include in a file prologue.

3. When adding a comment to a variable declaration, always begin the comment one space after the end of the declaration. (T / F)

4. To get the most on each line of code, always break a long line at the point determined by your text editor or integrated development environment (IDE) (T / F)

5. For an if or while that has only one statement in its body, braces for the body are optional. The compiler does not require them, but proper style suggests that you should include them. Give at least one reason why it's a good idea to put braces around the body's single statement.

6. What's wrong with the style of a class description that ends like this?



```
    }
  }
}
```

What might you do to fix it?

7. What should you use to separate large "chunks" of code?

8. For each, write "yes" or "no" to indicate whether it is good style to include a blank space

- after the single asterisks in the prologue

- between a method call and its opening parentheses
- within each of the three components in a for loop header
- after the two semicolons in the for loop header
- between a closing brace and the //'s for its associated comment
- after the //'s for all comments
- after the if, while, and switch keywords

§8.4 Helper Methods

9. Which of the following is a legitimate reason for creating a helper method?

a) You want the method to be hidden from the outside world.

b) You have a long and complicated method and would like to partition it into several smaller modules.

c) Your class contains two or more methods where some of the code is the same in both methods.

d) All of above.

10. Does a class's interface include the names of private methods?

§8.5 Encapsulation (with Instance Variables and Local Variables)

11. In the interest of encapsulation, use local variables instead of instance variables whenever possible. (T / F)

12. If a method modifies a particular instance variable, and if a program calls the same method two separate times, the value of the instance variable at the beginning of the second method call is guaranteed to be the same as the value it had at the end of the first method call. (T / F)

§8.7 Design Philosophy

13. Because some of your preliminary code might change in the course of development, do not waste time testing until everything is done. (T / F)

14. When you are testing a program, it's important to not have any preconceived expectations of what your output should look like. (T / F)

§8.8 Top-Down Design

15. The top-down design methodology is good because:

- **a)** It keeps everyone focused on a common goal. (T / F)
- **b)** It avoids "reinventing the

wheel.” (T / F)

c) It keeps management informed. (T / F)

d) It minimizes the chances of solving the wrong problem. (T / F)

e) It minimizes overall cost. (T / F)

f) It results in the fewest number of undetected bugs. (T / F)

16. In a top-down design process, which do you decide on first—the classes or the public methods?

§8.9 Bottom-Up Design

Page 377

17. When should you use bottom-up design?

§8.11 Iterative Enhancement

18. If a prototype is successful, what temptation should you resist?

19. Once you select a particular design methodology, keep using that same methodology throughout the entire design process, and do not allow other methodologies to “contaminate” the process originally selected. (T / F)

§8.12 Merging Driver Method into Driven Class

20. You can drive any class from a main method within that class, and you can retain that main method for future testing of that class even though that class is normally driven from another class in a larger program. (T / F)

§8.14 Writing Your Own Utility Class

21. A utility class’s members

normally should use the private and static modifiers. (T / F)

Exercises

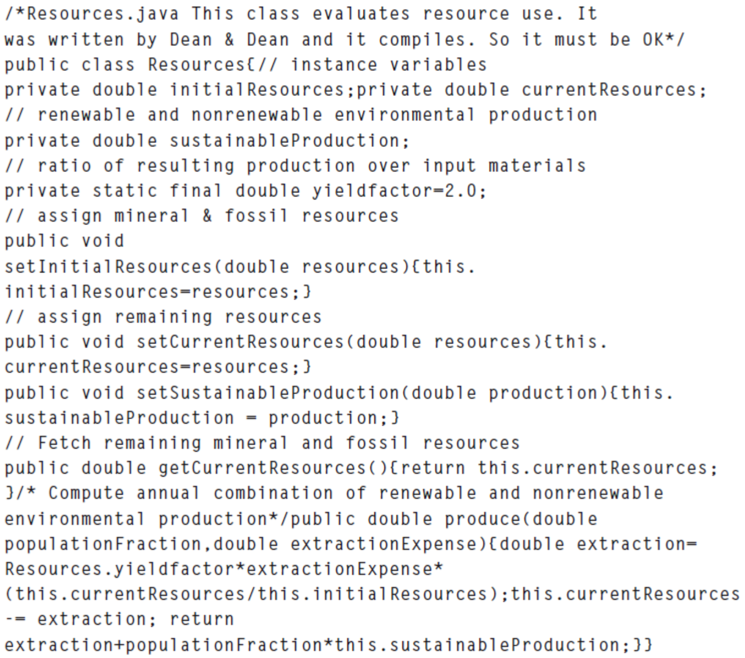
---

1. [after §8.2] Evaluate these assertions about styling of variable declarations and associated comments:

a) Defer all variable declaration until immediately before first use. (T / F)

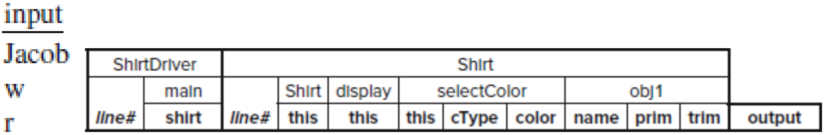
b) Use comments to describe variables whose meanings are not obvious. (T / F)

2. [after §8.2] Correct the style of the following program.



```
/*Resources.java This class evaluates resource use. It
was written by Dean & Dean and it compiles. So it must be OK*/
public class Resources{// instance variables
private double initialResources;private double currentResources;
// renewable and nonrenewable environmental production
private double sustainableProduction;
// ratio of resulting production over input materials
private static final double yieldfactor=2.0;
// assign mineral & fossil resources
public void
setInitialResources(double resources){this.
initialResources=resources;}
// assign remaining resources
public void setCurrentResources(double resources){this.
currentResources=resources;}
public void setSustainableProduction(double production){this.
sustainableProduction = production;}
// Fetch remaining mineral and fossil resources
public double getCurrentResources(){return this.currentResources;
}/* Compute annual combination of renewable and nonrenewable
environmental production*/public double produce(double
populationFraction,double extractionExpense){double extraction=
Resources.yieldfactor*extractionExpense*
(this.currentResources/this.initialResources);this.currentResources
-= extraction; return
extraction+populationFraction*this.sustainableProduction;}}
```

3. [after §8.4] Trace the Shirt program in Figures 8.6, 8.7a, and 8.7b, using either the short form or the long form. To help you get started, here's the trace setup, including the input. If you use the long form, you'll need to add line numbers to the code shown in Figures 8.6, 8.7a, and 8.7b. Do not omit line numbers for the blank lines. For example, the blank line above the selectColor method is line 41. If you use the short form, line numbers are unnecessary, and the *line#* heading shown below is unnecessary.



input
Jacob
w
r

ShirtDriver		Shirt									
	main		Shirt	display	selectColor			obj1			
line#	shirt	*line#*	this	this	this	cType	color	name	prim	trim	output

4. [after §8.5] This exercise demonstrates using a reference parameter to pass data back to the argument supplied in the method call. Suppose you want a Car5 class to include a method with this heading:

public int copyTo(Car5 newCar)

This method is supposed to be called by an existing Car5 object with another Car5 object as the argument. When any instance variable of the calling car has been initialized and the corresponding variable of the parameter car has not been initialized, the method should assign that calling-car variable value to the corresponding parameter-car variable and increment a counter that keeps track of the number of features copied. After trying to copy each of the calling car's instance variable values, the method should return the total number values copied. Here's a driver that illustrates the usage:

```
/*************************************************************
* Car5Driver.java
* Dean & Dean
*
* This class exercises the Car5 class.
*************************************************************/

public class Car5Driver
{
  public static void main(String[] args)
  {
    Car5 marcusCar = new Car5();
    Car5 michaelCar = new Car5();

    System.out.println(marcusCar.copyTo(michaelCar) +
      " features copied");
    marcusCar = new Car5("Chevrolet", 2020, "blue");
    System.out.println(marcusCar.copyTo(michaelCar) +
      " features copied");
  } // end main
} // end class Car5Driver
```

Output:

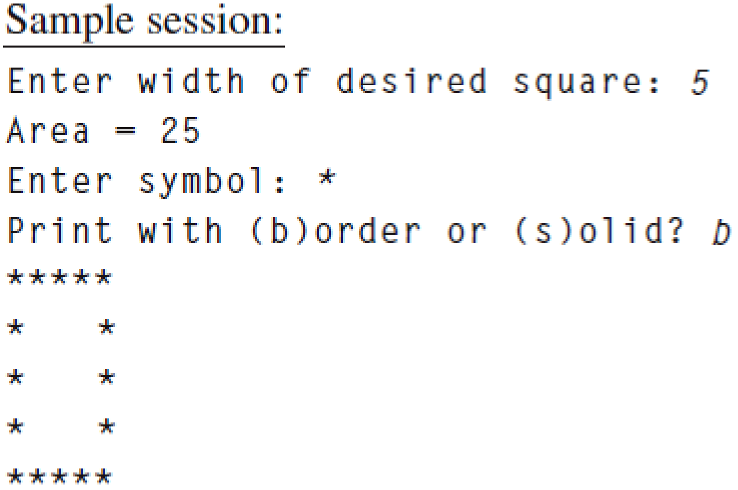
```
0 features copied
3 features copied
```

Write the code for the desired copyTo method.

5. [after §8.7] Whenever a programmer is asked to enhance an existing program, there is risk of introducing a bug that breaks something that was not broken in the original program.

- **a)** What should the original programmer do to help a successor understand the complexities in the original program?
- **b)** What should the original programmer do to help a successor verify that unintentional bugs are not accidentally introduced into satisfactory parts of the program?

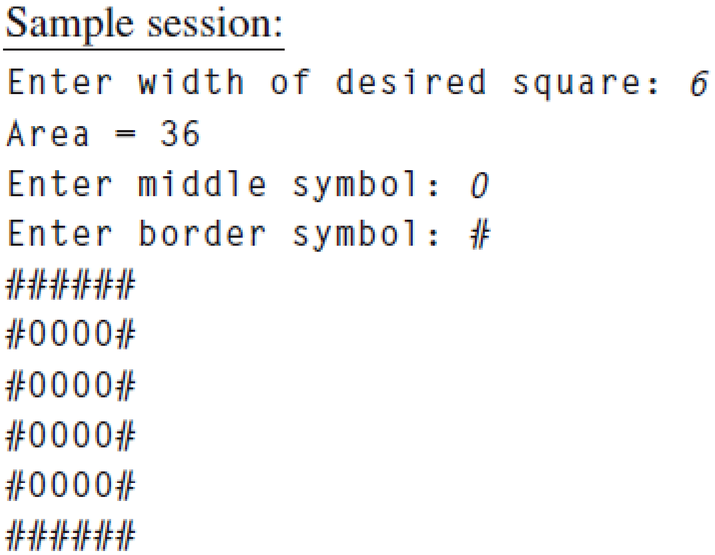
6. [after §8.8] Modify the Square class in Figures 8.13a and 8.13b so that it uses an instance variable to hold the printed symbol for either the border square or the solid square.



Sample session:

```
Enter width of desired square: 5
Area = 25
Enter symbol: *
Print with (b)order or (s)olid? b
*****
*   *
*   *
*   *
*****
```

7. [after §8.8] Modify the draw method in the Square class in Figure 8.13a. Instead of prompting the user for border or solid, the draw method should prompt the user for an interior character and a border character. If the user enters a space for the interior character, the result will look like a border square. If the user enters the same character for both the interior and the border, the result will look like a solid square. If the user enters different characters, the border will look different from the interior. Your new draw method must do all the work of the original draw, drawBorderSquare, drawSolidSquare, drawHorizontalLine, and drawSides methods in Figures 8.13a and 8.13b. That means the drawBorderSquare, drawSolidSquare, drawHorizontalLine, and drawSides methods will be unnecessary.



Sample session:

```
Enter width of desired square: 6
Area = 36
Enter middle symbol: 0
Enter border symbol: #
######
#0000#
#0000#
#0000#
#0000#
######
```

8. [after §8.8] Figure 8.2b has two if statement conditions that contain what are called regular expressions. As indicated, these are explained in the Java API Pattern class. This exercise is intended to help you get a better feeling for Java’s regular expressions and their usage. Use your Java API documentation on the Pattern class to answer these questions:

a) Write the regular expression for a character string starting with a ‘Z’ and containing any number of additional characters of any kind except for a space or a tab.

b) Write the regular expression for a string that represents a U.S. long-distance telephone number (three digits, a hyphen or space, three digits, a hyphen or space, and four digits).

c) What is the meaning of the regular expression, "[A–Z][a–z]*", which appears in Figure 8.2b?

9. [after §8.8] Top-Down Design:

a) Helps managers control development of large software projects. (T / F)

b) Maximizes use of prewritten software. (T / F)

10. [after §8.8] Write stubs for all the constructors and methods in the Student class of Figures 8.2a and 8.2b. Each stub should print the constructor or method name followed by the passed-in values of all parameters. A variation of the driver in Figure 8.1 might produce this output:

```
in Student
in Student, first = Jill, last = McDonald
in setFirst, first = Hyoung
in setLast, last = Jhang
in printFullName
```

11. [after §8.9] Top-down design uses stubs to represent not-yet-implemented methods. Bottom-up design uses local main methods to test low-level methods that will eventually be called from other classes. The public method below draws a solid right triangle. Provide a local main method to test it.

```
public class Triangle
{
  public void draw(double height, double tanApex)
  {
    int rows = (int) Math.ceil(height);
    int rowLength;

    for (int row=1; row<=rows; row++)
    {
      rowLength = (int) Math.ceil(row * tanApex);
      for (int j=0; j<rowLength; j++)
      {
        System.out.print("*");
      }
      System.out.println();
    }
  } // end draw
} // end Triangle
```

Sample session:

```
Triangle height = 5
Tangent of apex angle = 0.9
*
**
***
****
*****
```

12. [after §8.11] Write a prototype of the Square program, using just one class named SquarePrototype, and using only one method, main. Your program should generate the prescribed border output for the case of when the user selects the (b)order option. If the user selects the (s)olid option, your program should display "Not Implemented."

13. [after §8.11] When you design software, you should always use the top-down methodology. (T / F)

14. [after §8.12] Given the existing Time class shown in Figure 8.14. Implement a TimeDriver class with a main method that (1) prompts the user for hours and minutes values and (2) calls the Time constructor with those input values for the hours and minutes arguments, and with 0 for the seconds argument.

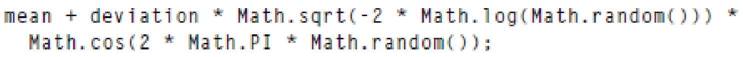
15. [after §8.13] For the Square class in Figure 8.13a, rewrite the Square constructor and the getArea method so that there's no this prefix.

16. [after §8.14] Implement a utility class named Distribution, which contains the following two static methods.

a) The exponential method returns an exponential distribution value calculated with this formula, where mean is a double parameter which holds the distribution's mean:

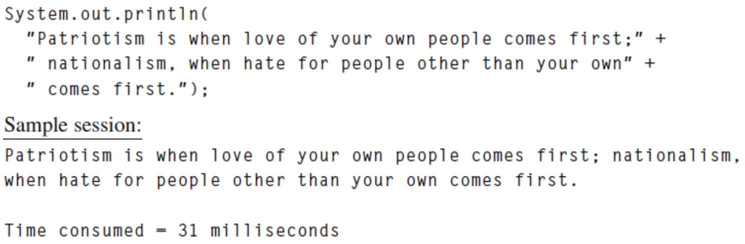
-Math.log(Math.random()) * mean

b) The normal method returns a normal distribution value calculated with this formula, where mean and deviation are double parameters that hold the distribution's mean and standard deviation, respectively:



```
mean + deviation * Math.sqrt(-2 * Math.log(Math.random())) *
  Math.cos(2 * Math.PI * Math.random());
```

17. [after §8.15] The Java API Calendar class contains a method called getTimeInMillisec, which enables you to retrieve the absolute time (in milliseconds) at which any Calendar object was created. You can use this capability to measure the runtime of any chunk of code. All you have to do is create a Calendar object before the test code starts, create another Calendar object right after the code ends, and print out the difference in those two objects' times. As indicated in Section 8.15, you can create a Calendar object by calling the Calendar class's static method, getInstance. For this exercise, you need to implement a class named MeasureRuntime that measures the runtime for the following statement (which displays a Charles de Gaulle quote):



```
System.out.println(
  "Patriotism is when love of your own people comes first;" +
  " nationalism, when hate for people other than your own" +
  " comes first.");
```

Sample session:

```
Patriotism is when love of your own people comes first; nationalism,
when hate for people other than your own comes first.

Time consumed = 31 milliseconds
```

Review Question Solutions

---

1. False. Readability is an important attribute of good computer code. To save printer paper, print on both sides of the page and/or use smaller font.

2. The seven items to include in a file prologue are:

- **line of asterisks**
- **filename**
- **programmer name(s)**
- **blank line with one asterisk**
- **description**
- **line of asterisks**
- **blank line**

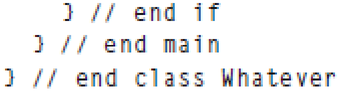
3. False. That would provide maximum room for each comment, but good programmers make the beginnings of declaration comments line up with each other, and they try to make declaration comments short enough to avoid line wrap.

4. False. Take control, and break a long line at the most logical place(s).

5. Even though it's not necessary, it's a good idea to provide braces with single-statement if and while statements because

- **Braces provide a visual cue for remembering to indent.**
- **Braces help you avoid a logical mistake if you add code later.**

6. Unless a block is very short, it may not be immediately obvious which block is being terminated by a particular brace. It’s good practice to terminate all but the shortest blocks with a comment, for example,



```
    } // end if
  } // end main
} // end class Whatever
```

7. Separate large chunks of code with blank lines.

8. Yes means include a space, No means do not.

- Yes, after the single asterisks in the prologue.
- No, not between a method call and its opening parentheses.
- No, not within each of the three components in a for loop header.
- Yes, after the two semicolons in the for loop header.
- Yes, between a closing brace and the //’s for its associated comment.
- Yes, after the //’s for all

comments.

- Yes, after the if, while, and switch keywords.

9. d) All of the above.

10. No. the interface does not describe private members.

11. True. You should generally try to keep things as local as possible, and using local variables instead of instance variables is one way to do this. Instance variables should be reserved for attributes that describe an object's state.

12. False. It's true that an instance variable persists throughout the life of an object, and if the second call of the same method were right after the first call of that method, the final value of the instance variable in the first call would be the same as the initial value in the second call of that method. However, it's possible that some other method could change the value of the instance variable between the two calls of the method in question.

13. False. Test frequently throughout the development process.

14. False. It's important to have a clear idea of what you expect to see before you do a test, so you will have the best chance of recognizing a discrepancy when it occurs.

15. The top-down design methodology is the best because:

a) True.

b) False. It sometimes forces people to re-invent the wheel.

c) True

d) False. If you're worried about solving the wrong problem, use prototyping.

e) False. To minimize cost, organize the design to reuse existing components.

f) False. To maximize reliability, organize the design to reuse existing components.

16. In top-down design you decide on the classes before the public methods.

17. You should use bottom-up design when your program can utilize a substantial amount of prewritten software or when low-level details are critical and require early attention.

18. If a prototype is successful, it’s important to resist the temptation of continuing the development by tinkering with that prototype.

19. False. Many problems need benefits of more than one design methodology. It's a good idea to stick with one methodology through one design cycle (planning, implementation, testing and evaluation), but you might need to switch to a different methodology in the next design iteration.

20. True. The particular main method used is the one that's current when execution starts.

21. False. A utility class's members should normally use the public and static modifiers.

---

[1] Institute of Electrical and Electronics Engineers (IEEE) Standard 610.12, https://standards.ieee.org/standard/24765-2017.html.

[2] "Java Code Conventions," September 12, 1997, http://www.oracle.com/technetwork/java/codeconventions-150003.pdf.

[3] For other options and other arguments, enter javadoc -help.

[4] http://msdn.microsoft.com/en-us/library/ff649643.aspx. See https://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

[5] Of course, we, John and Ray, never make any mistakes. ☺

[6] If the condition is false, the program throws an AssertionError exception and terminates execution. You’ll learn about exceptions later in this book, in Chapter 15.

[7] Although we have been encouraging you to think of Java API software as being completely encapsulated, Oracle does not keep the Java API source code secret. It can be downloaded and is available for inspection by Java developers.

[8] The set of commands you can enter

into a Windows command prompt window are essentially DOS commands—they are a software legacy of the IBM PC that came out in the early 1980s. All of the original DOS fit on a single 5.25″ diameter, 128-kilobyte floppy disk.

[9] In principle, it's possible to build software systems out of components that are commercial-off-the-shelf (COTS) programs from different companies. This strategy can be used to avoid "reinventing the wheel" in a big way, and it minimizes new code to the "glue" that provides component interfaces. However, it takes longer to write this glue code than it does to write ordinary code. Moreover, because (in general) the system developer does not have access to component source code and does not have control of component evolution, the development process is relatively risky, and the resulting composite

program is relatively brittle. COTS-based system design has a distinctive methodology that is outside the scope of this text.

[10] See https://www.fsf.org. The Free Software Foundation is “dedicated to promoting computer users’ rights to use, study, copy, modify, and redistribute computer programs.” Two famous examples of this kind of software are the GNU/Linux operating system (GNU stands for “Gnu’s Not Unix”) and the Apache software that underlies most web servers (https://www.apache.org).

[11] OOPSLA ’89 Conference Proceedings: Object-Oriented Programming Systems, Languages and Applications (New Orleans, Louisiana, October 1–6, 1989).

Arrays

Objectives

- Compare an array to other objects.
- Create and initialize arrays.
- Copy values from one array to another.
- Shift data in an array.

- Make histograms.
- Search an array for particular data.
- Sort data.
- Create and use two-dimensional arrays.
- Create and use arrays of objects.
- Learn how to use for-each loops.

Outline

9.1 Introduction 385

9.2 Array Basics 385

9.3 Array Declaration and Creation 387

9.4 Array length Property and Partially Filled Arrays 391

9.5 Copying an Array 393

9.6 Problem Solving with Array

Case Studies 397

9.7 Searching an Array 403

9.8 Sorting an Array 408

9.9 Two-Dimensional Arrays 412

9.10 Arrays of Objects 418

9.11 For-Each Loops 425

9.1 Introduction

In the past, you've seen that objects typically contain more than one data item, and the different data items each have a different name. Now, we'll look at a special kind of object that holds several items of the same type and uses the same name for all of them. Natural language has ways to give a single name to a population: "pack" of wolves, "herd" of cattle, "pride" of lions, “passel” of possums, “fesnying” of ferrets, and so on. Java has a way to do the same thing.

When you have a collection of items of the same type, and you’d like to use the same name for all of them, you can define them all together as an *array*. Each item in the array is more formally called an array *element*. To distinguish the different elements in the array, you use the array name plus a number that identifies the position of the element within the array. For example, if you stored a collection of song titles in an array named songs, you’d distinguish the first song title by saying songs[0], and you’d distinguish the second song title by saying songs[1]. As evidenced by this example, array elements start at position 0. An array’s position numbers (0, 1, 2, etc.) are more formally called *indexes*. We’ll have more to say about array indexes in the next section.

There’s an important advantage in

using one name for all of a group of similar items and distinguishing them only by a number. It can lead to simpler code. For example, if you need to store 100 song titles, you could declare 100 separate variables. But what a pain it would be to have to write 100 declaration statements and keep track of 100 different variable names. The easier solution is to use an array and declare just one variable—a songs array variable.🚲

Readers who want an early introduction to arrays have the option of reading Sections 9.1 through 9.6 after completing Chapter 4. The natural connection between Chapter 4 and this chapter is that Chapter 4 describes loops and arrays rely heavily on loops.📖

Starting with Section 9.7, we present arrays in an object-oriented context. More specifically, we use constructors, instance methods, and static methods for the programs that illustrate the new array concepts. We discuss techniques for searching an array and sorting an array. We describe different organizational structures for arrays—two-dimensional arrays and arrays of objects. Finally, we describe a special type of for loop called a for-each loop, which allows you to loop through the elements in an array without having to declare or use an index variable.

9.2 Array Basics

In this section, we show you how to perform simple operations on an array, such as loading an array with data and printing an array. To illustrate these operations, we'll refer to the phoneList array in Figure 9.1. The phoneList array holds a list of five contact phone numbers for a cell phone. The first phone number is 8167412000, the second phone number is 2024561111, and so on.

Accessing an Array's Elements

To work with an array, you need to access an array's elements. For example, to print the contents of an array, you need to access the array's first element, print it, access the array's second element, print it, and so on. To access an element within an array, you specify the array's name, followed by square brackets surrounding the element's index. Figure 9.2 shows how to access the individual elements within the phoneList array. The first element's index is 0, so you access the first element with phoneList[0]. Why is the first element's index 0 instead of 1? The index is a measure of how far you are from the beginning of the array. If you're right at the beginning, the distance from the beginning is 0. So the first element uses 0 for its index value.

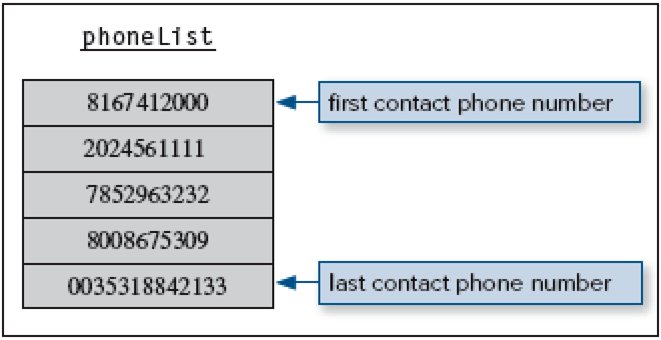

phoneList
8167412000
2024561111
7852963232
8008675309
0035318842133
first contact phone number
last contact phone number

Figure 9.1 Example array—five-element array for holding a list of contact phone numbers

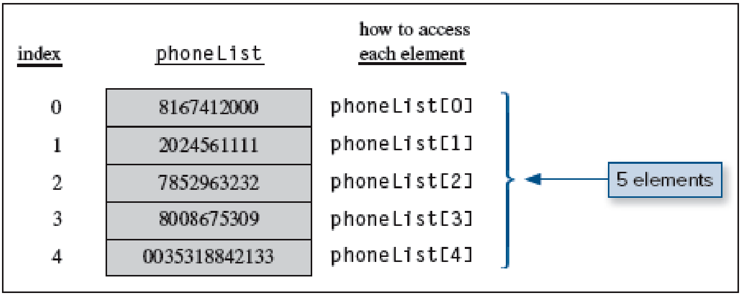

index
phoneList
how to access each element
0
1
2
3
4
8167412000
2024561111
7852963232
8008675309
0035318842133
phoneList[0]
phoneList[1]
phoneList[2]
phoneList[3]
phoneList[4]
5 elements

Figure 9.2 Accessing elements in a phoneList array

Page 386

⚠ Beginning programmers often think that the last index in an array is equal to the number of elements in the array. For example, a beginning programmer might think that the last index in the phoneList array equals 5 because the phoneList array has 5 elements. Not so. The first index is 0, and the last index is 4. Try to remember this important rule: The last index in an array is equal to one less than the number of elements in the array. If you attempt to access an array element with an index that's greater than the last index or less than zero, you'll get a program crash. So if you specify phoneList[5] or phoneList[-1], you'll get a program crash. As part of that crash, the Java virtual machine prints an error message with the word "ArrayIndexOutOfBoundsException" in it. ArrayIndexOutOfBoundsException is an *exception*. You'll learn about exceptions in Chapter 15, but for now, just think of an exception as a sophisticated type of error that can be used by programmers to determine the source of a bug.

Now that you know how to access an array element, let's put it to use. Here's how you can change the first phone number to 2013434:

```
phoneList[0] = 2013434;
```

And here's how you can print the second phone number:

```
System.out.println(phoneList[1]);
```

Be aware that some people use the term "subscript" rather than "index" because subscripting is the standard English way to represent an element from within a group. In other words, x0, x1, x2, and so on in ordinary writing is the same as x[0], x[1], x[2], and so on in Java.

Example Program

Let's see how arrays are used within the context of a complete program. In Figure 9.3, the ContactList program prompts the user for the number of contact phone numbers that are to be entered, fills up the phoneList array with user-entered phone numbers, and prints the created contact list. To fill an array and to print an array’s elements, you typically need to step through each element of the array with the help of an index variable that increments from zero to the index of the array’s last filled element. Often, the index variable’s increment operations are implemented with the help of a for loop. For example, the ContactList program uses the following for loop header to increment an index variable, i:

```
for (int i=0; i<sizeOfList; i++)
```

With each iteration of the for loop, i goes from 0 to 1 to 2, and so on, and i serves as an index for the different elements in the phoneList array. Here’s how the loop puts a phone number into each element:

```
phoneList[i] = phoneNum;
```

9.3 Array Declaration and Creation

In the previous section, we showed you how to perform simple operations on an array. In so doing, we focused on accessing an array's elements. In this section, we focus on another key concept—declaring and creating arrays.

Array Declaration

An array is a variable and, as such, you must declare it before you can use it. To declare an array, use this syntax:

element-type[] *array-variable*;

The *array-variable* is the name of the array. The empty square brackets tell us that the variable is defined to be an array. The *element-type* indicates the type of each element in the array—int, double, char, String, and so on.

Here are some array declaration examples:

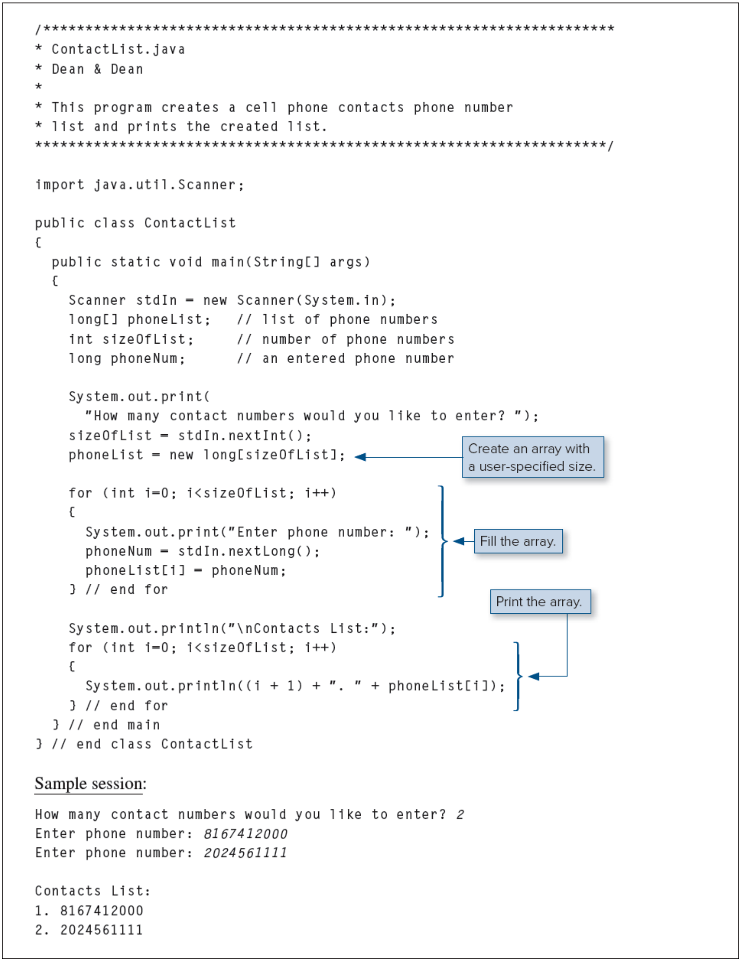
```
double[] salaries;
String[] names;
int[] employeeIds;
```

The salaries variable is an array whose elements are of type double. The names variable is an array whose elements are of type String. And finally, the employeeIds variable is an array whose elements are of type int.

Java provides an alternative declaration format for arrays, where the square brackets go after the variable name. Here's what we're talking about:

```
double salaries[];
```

The two formats are identical in terms of functionality. Most folks in industry prefer the first format, and that's what we use, but you should be aware of the alternative format in case you see it in someone else's code.



```
/*********************************************************************
* ContactList.java
* Dean & Dean
*
* This program creates a cell phone contacts phone number
* list and prints the created list.
*********************************************************************/

import java.util.Scanner;

public class ContactList
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    long[] phoneList;   // list of phone numbers
    int sizeOfList;     // number of phone numbers
    long phoneNum;      // an entered phone number

    System.out.print(
      "How many contact numbers would you like to enter? ");
    sizeOfList = stdIn.nextInt();
    phoneList = new long[sizeOfList];

    for (int i=0; i<sizeOfList; i++)
    {
      System.out.print("Enter phone number: ");
      phoneNum = stdIn.nextLong();
      phoneList[i] = phoneNum;
    } // end for

    System.out.println("\nContacts List:");
    for (int i=0; i<sizeOfList; i++)
    {
      System.out.println((i + 1) + ". " + phoneList[i]);
    } // end for
  } // end main
} // end class ContactList
```

Create an array with a user-specified size.

Fill the array.

Print the array.

Sample session:

```
How many contact numbers would you like to enter? 2
Enter phone number: 8167412000
Enter phone number: 2024561111

Contacts List:
1. 8167412000
2. 2024561111
```

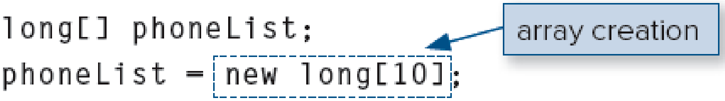
Figure 9.3 ContactList program that shows how to create, fill, and print an array

Array Creation

An array is an object, albeit a special kind of object. As with any object, an array holds a group of data items. As with any object, an array can be created/instantiated using the new operator. Here's the syntax for creating an array object with the new operator and assigning the array object into an array variable:

array-variable = new *element-type*[*array-size*];

The *element-type* indicates the type of each element in the array. The *array-size* indicates the number of elements in the array. The following code fragment creates a 10-element array of longs:



```
long[] phoneList;
phoneList = new long[10];
```

array creation

These two lines perform three operations: (1) The first line declares the phoneList variable, (2) the boxed code creates the array object, and (3) the assignment operator assigns a reference to the array object into the phoneList variable.

It’s legal to combine an array’s declaration, creation, and assignment operations into one statement. The following example does just that. It reduces the previous two-line code fragment to just one line:

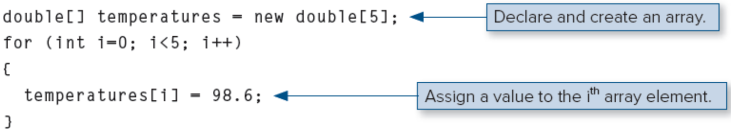
```
long[] phoneList = new long[10];
```

Here, we use a constant (10) for the array’s size, but you’re not required to use a constant. You can use any expression for the array’s size. Figure 9.3’s ContactList program prompts the user for the size of the array, stores the entered size in a sizeOfList variable, and uses sizeOfList for the array creation. Here’s the array creation code from the ContactList program:

```
phoneList = new long[sizeOfList];
```

Array Element Initialization

Usually, you'll want to declare and create an array in one place and assign values to your array elements in a separate place. For example, the following code fragment declares and creates a temperatures array in one statement, and assigns values to the temperatures array in a separate statement, inside a loop.



```
double[] temperatures = new double[5];
for (int i=0; i<5; i++)
{
  temperatures[i] = 98.6;
}
```

Declare and create an array.

Assign a value to the ith array element.

On the other hand, sometimes you'll want to declare and create an array and assign values to your array, all in the same statement. That's called an *array initializer*. Here's the syntax:

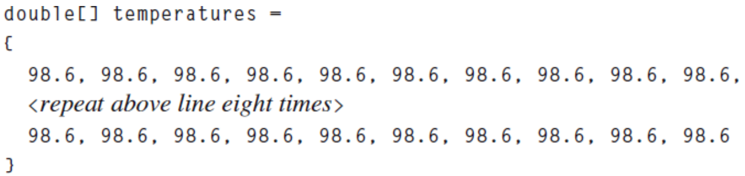
> *element-type*[] *array-variable* = {*value1*, *value2*, . . ., *valuen*};

The code at the left of the assignment operator declares an array variable using syntax that you've seen before. The code at the right of the assignment operator specifies a comma-separated list of values that are assigned to the array's elements. Note this example:

```
double[] temperatures = {98.6, 98.6, 98.6, 98.6, 98.6};
```

Comparing the above statement to the previous temperatures code fragment, you can see that it is the same in terms of functionality but different in terms of structure. Key differences: (1) It's one line, rather than five lines. (2) There's no new operator. (3) There's no array-size value. With no array-size value, how do you think the compiler knows the size of the array? The size of the array is dictated by the number of values in the element-values list. In the above example, there are five values in the initializer list, so the compiler creates an array with five elements.

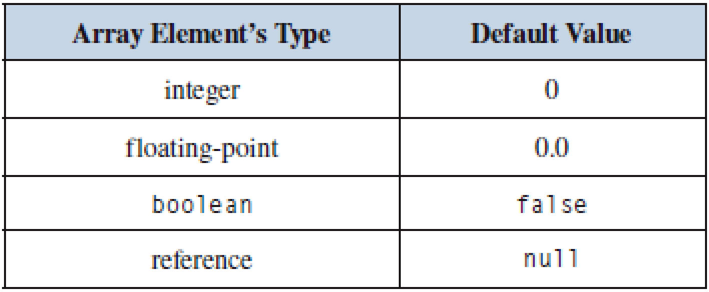
We presented two solutions for assigning values to a temperatures array. Which is better—the five-line code fragment or the one-line array initializer? We prefer the array initializer solution because it's simpler. But remember that you can use the array initializer technique only if you know the assigned values when you first declare the array. For the temperatures example, we do know the assigned values when we first declare the array—we initialize each temperature to 98.6, the normal human body temperature in degrees Fahrenheit. You should limit your use of array initializers to situations where the number of assigned values is reasonably small. For the temperatures example, the number of assigned values is reasonably small—it's five. If you need to keep track of a hundred temperatures, it would be legal to use the array initializer solution, but it would be cumbersome:



```
double[] temperatures =
{
  98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6,
  <repeat above line eight times>
  98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6, 98.6
}
```

Default Values

You now know how to initialize an array's elements explicitly with an array initializer. But what do an array's elements get by default if you don't use an array initializer? An array is an object, and an array's elements are the instance variables for an array object. As such, an array's elements get default values when the array is created, just as any other instance variables get default values. Here are the default values for array elements:



Array Element's Type	Default Value
integer	0
floating-point	0.0
`boolean`	`false`
reference	`null`

So what are the default values for the

elements in the arrays below?

```
double[] rainfall = new double[365];
String[] colors = new String[5];
```

The rainfall array gets 0.0 for each of its 365 elements. The colors array gets null for each of its 5 elements.

9.4 Array length Property and Partially Filled Arrays

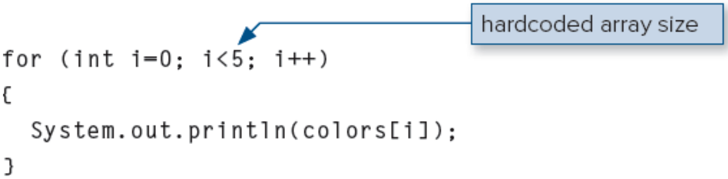
As illustrated earlier, when working with an array, it’s common to step through each element in the array. In doing so, you need to know the size of the array and/or the number of filled elements in the array. In this section, we discuss how to obtain the size of an array and how to keep track of the number of filled elements in an array.

Array length Property

Suppose you have a five-element colors array that's been initialized like this:

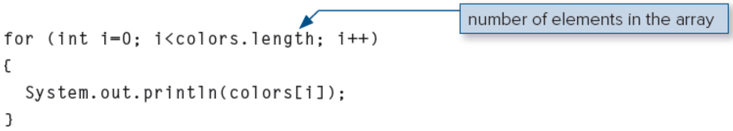
String[] colors = {"blue", "gray", "lime", "teal", "yellow"};

Here's how to print such an array:



```
for (int i=0; i<5; i++)
{
  System.out.println(colors[i]);
}
```

That works OK, but suppose that you have several other color-related loops in your code, each of them using i<5. If you modify your program to accommodate more colors, and change the 5-element array to a 10-element array, you'd have to change all occurrences of i<5 to i<10. To avoid such maintenance work, wouldn't it be nice to replace i<5 or i<10 with something generic, like i < array's size? You can do that by using the color array's length property. Every array object contains a length property that stores the number of elements in the array. The length property is called a “property,” but it’s actually just an instance variable with public and final modifiers. The public modifier says that length is directly accessible without need of an accessor method. The final modifier makes length a named constant; so you can’t update it. Here’s how the length property can be used:

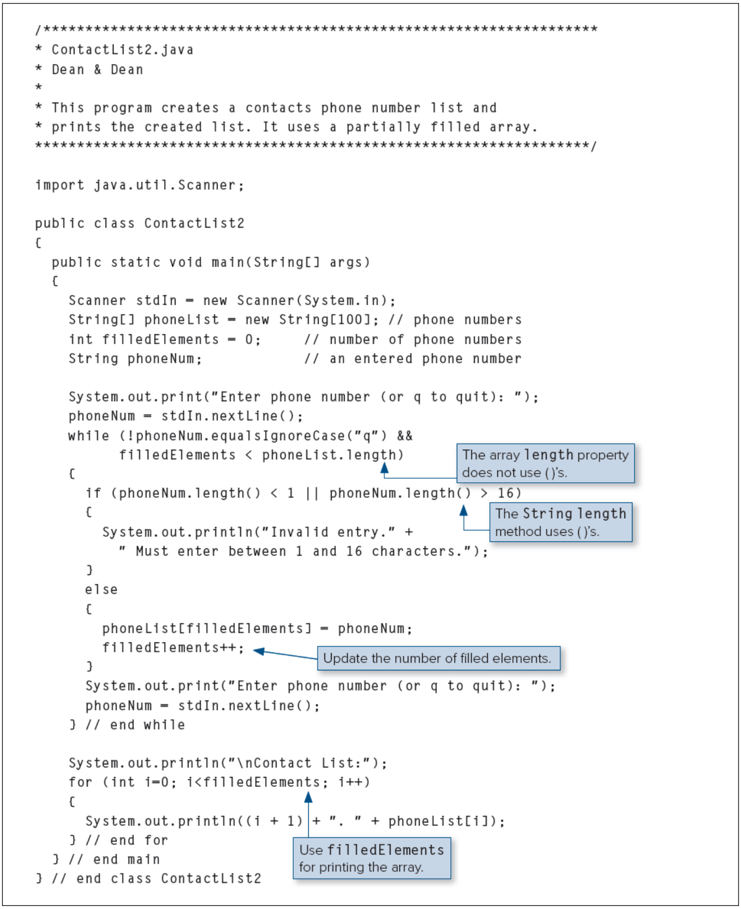


```
for (int i=0; i<colors.length; i++)
{
  System.out.println(colors[i]);
}
```

Array length Property Versus String length Method

Remember where else you’ve seen the word length in the Java language? The String class provides a length method to retrieve the number of characters in a string. Remember that String’s length is a method, so you must use trailing parentheses when calling it. On the other hand, an array's length is a constant, so you don't use trailing parentheses when accessing it. Figure 9.4's ContactList2 program illustrates these concepts. Note that phoneNum.length() uses parentheses when checking for the length of the phoneNum string as part of input validation. And note that phoneList.length does not use parentheses when checking the number of elements in the phoneList array to make sure that there's room for another phone number.

If you're like us, you might have a hard time remembering when to use parentheses and when not to. Try using the mnemonic acronym ANSY, which stands for Arrays No, Strings Yes. "Arrays No" means that arrays do not use parentheses when specifying length. "Strings Yes" means that strings do use parentheses when specifying length. If you don’t like DFLAs,[1] you can try a more analytical approach to remembering the parentheses rule. Arrays are special-case objects that don’t have methods; therefore, an array’s length must be a constant, not a method. And constants don’t use parentheses.



```
/*******************************************************************
* ContactList2.java
* Dean & Dean
*
* This program creates a contacts phone number list and
* prints the created list. It uses a partially filled array.
*******************************************************************/

import java.util.Scanner;

public class ContactList2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String[] phoneList = new String[100]; // phone numbers
    int filledElements = 0;      // number of phone numbers
    String phoneNum;             // an entered phone number

    System.out.print("Enter phone number (or q to quit): ");
    phoneNum = stdIn.nextLine();
    while (!phoneNum.equalsIgnoreCase("q") &&
          filledElements < phoneList.length)
    {
      if (phoneNum.length() < 1 || phoneNum.length() > 16)
      {
        System.out.println("Invalid entry." +
          " Must enter between 1 and 16 characters.");
      }
      else
      {
        phoneList[filledElements] = phoneNum;
        filledElements++;
      }
      System.out.print("Enter phone number (or q to quit): ");
      phoneNum = stdIn.nextLine();
    } // end while

    System.out.println("\nContact List:");
    for (int i=0; i<filledElements; i++)
    {
      System.out.println((i + 1) + ". " + phoneList[i]);
    } // end for
  } // end main
} // end class ContactList2
```

The array `length` property does not use ()'s.

The `String length` method uses ()'s.

Update the number of filled elements.

Use `filledElements` for printing the array.

Figure 9.4 ContactList2 program This processes a partially filled array, using the array length property and the String length method.

Partially Filled Arrays

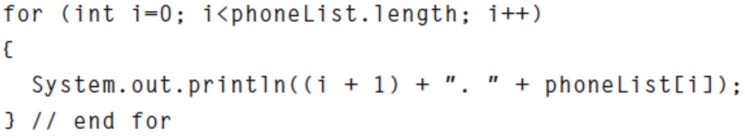
In Figure 9.4, note how the

ContactList2 program declares the phoneList array to have 100 elements. The program repeatedly prompts the user to enter a phone number or enter q to quit. Typically, the user will enter fewer than the maximum 100 phone numbers. That results in the phoneList array being partially filled. If you have a partially filled array, as opposed to a completely filled array, you have to keep track of the number of filled elements in the array so you can process the filled elements differently from the unfilled elements. Note how the ContactList2 program uses the filledElements variable to keep track of the number of phone numbers in the array. filledElements starts at zero and gets incremented each time the program stores a phone number in the array. To print the array, the program uses filledElements in the following for loop header:

```
for (int i=0; i<filledElements; i+
```

+)

It's fairly common for programmers to accidentally access unfilled elements in a partially filled array. ⚠ For example, suppose ContactList2's for loop looked like this:



```
for (int i=0; i<phoneList.length; i++)
{
  System.out.println((i + 1) + ". " + phoneList[i]);
} // end for
```

Using phoneList.length in the for loop header works great for printing a completely filled array, but not so great for printing a partially filled array. In the ContactList2 program, unfilled elements hold null (the default value for a string), so the above for loop would print null for each of the unfilled elements. And that makes for confused and unhappy users. ☺

9.5 Copying an Array

In the previous sections, we focused on array syntax details. In the next several sections, we'll focus less on the syntax and more on the application side of things. In this section, we discuss a general-purpose problem—how to copy from one array to another.

Using Arrays to Hold a Store's Prices

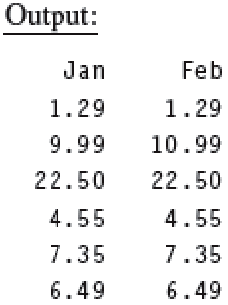
Suppose you use arrays to hold a store's prices, one array for each month's prices. Here's the array for January's prices:

```
double[] pricesJanuary = {1.29, 9.99, 22.50, 4.55, 7.35, 6.49};
```

Your intent is to use January's array as a starting point for the other month's arrays. Specifically, you want to copy January's prices into the other months' arrays and modify the other months' prices when necessary. The statement below creates the array for February's prices. Note how pricesJanuary.length ensures that February's array is the same length as January's array:

```
double[] pricesFebruary = new double[pricesJanuary.length];
```

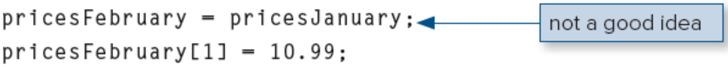
Suppose you want the values in February's array to be the same as the values in January's array except for the second entry, which you want to change from 9.99 to 10.99. In other words, you want something like this:



Output:

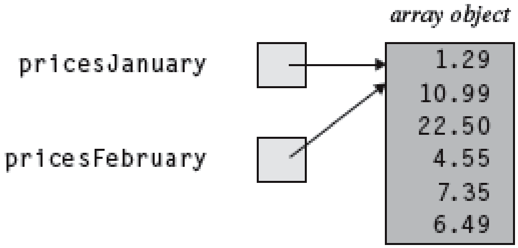
```
   Jan      Feb
  1.29     1.29
  9.99    10.99
 22.50    22.50
  4.55     4.55
  7.35     7.35
  6.49     6.49
```

To minimize re-entry effort and error, it would be nice to have the computer copy the first array's values into the second array and then just alter the one element of the second array that needs changing. Would the following code fragment work?



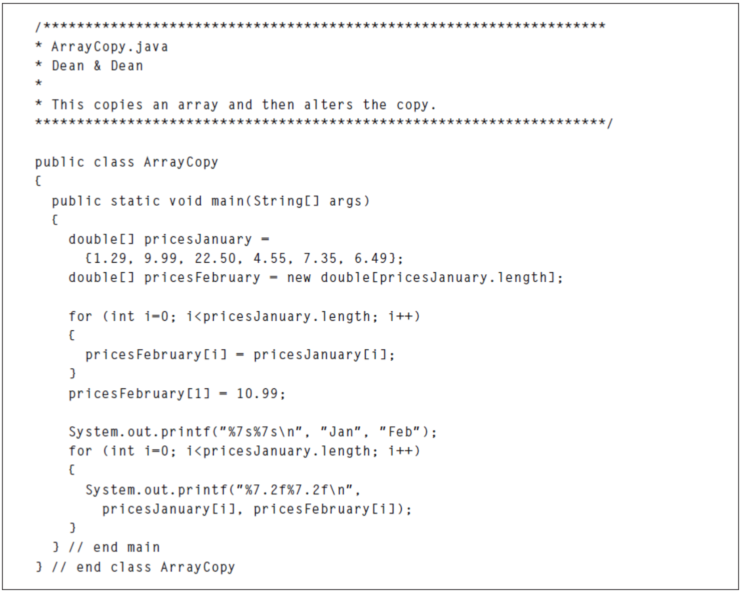
```
pricesFebruary = pricesJanuary;
pricesFebruary[1] = 10.99;
```

An array name is just a reference. It contains the address of a place in memory where the array's data begins. So pricesFebruary = pricesJanuary; gets the address of pricesJanuary's data and copies the ⚠ address into pricesFebruary. Then pricesFebruary and pricesJanuary refer to the same physical data. This picture illustrates the point:



The problem with pricesFebruary and pricesJanuary referring to the same physical data is that if you change the data for one of the arrays, then you automatically change the data for the other array. For example, the above pricesFebruary[1] = 10.99; statement updates not only pricesFebruary's second element, but also pricesJanuary's second element. And that's not what you want.

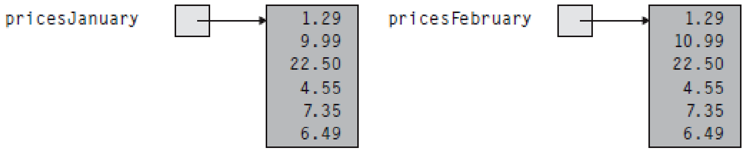
Usually when you make a copy of an array, you'll want the copy and the original to point to different array objects. To do that, assign array elements one at a time. See Figure 9.5's ArrayCopy program. It uses a for loop to assign pricesJanuary elements to pricesFebruary elements one at a time.



```
/*******************************************************************
* ArrayCopy.java
* Dean & Dean
*
* This copies an array and then alters the copy.
********************************************************************/

public class ArrayCopy
{
  public static void main(String[] args)
  {
    double[] pricesJanuary =
      {1.29, 9.99, 22.50, 4.55, 7.35, 6.49};
    double[] pricesFebruary = new double[pricesJanuary.length];

    for (int i=0; i<pricesJanuary.length; i++)
    {
      pricesFebruary[i] = pricesJanuary[i];
    }
    pricesFebruary[1] = 10.99;

    System.out.printf("%7s%7s\n", "Jan", "Feb");
    for (int i=0; i<pricesJanuary.length; i++)
    {
      System.out.printf("%7.2f%7.2f\n",
        pricesJanuary[i], pricesFebruary[i]);
    }
  } // end main
} // end class ArrayCopy
```

Figure 9.5 ArrayCopy program that copies an array and then alters the copy

This is what the code in Figure 9.5 produces:


pricesJanuary
1.29
9.99
22.50
4.55
7.35
6.49
pricesFebruary
1.29
10.99
22.50
4.55
7.35
6.49

Array Copying Methods Provided by Java's API

Copying data from one array to another is a very common operation, so Java designers provide several methods just for that purpose. The clone method comes free of charge with all objects, and it does what you'd expect; it instantiates a new version of the original object. To create a separate copy of an array, you can do this:

pricesFebruary = pricesJanuary.clone();

The System.arraycopy method

allows you to copy any number of elements from any place in one array to any place in another array. Here's how you could use it to copy Figure 9.5's pricesJanuary array to the pricesFebruary array:

```
System.arraycopy(pricesJanuary, 0, pricesFebruary, 0, 6);
```

Unlike the clone method, the arraycopy method does not instantiate a new object, so before calling it, you'll need to explicitly instantiate your target array. In the array method call, the first argument is the source array name, that is, the name of the array you're copying from. The second argument is the index of the source array's first element to copy. The third argument is the destination array name, that is, the name of the array you're copying to. The fourth argument is the index of the destination array's first element to replace. The final argument is the total number of elements to copy.

The Arrays class provides several additional array-copying methods—copyOf and copyOfRange. To access them, you'll need to include this import statement:

```
import java.util.Arrays;
```

As with the System.arraycopy method, the copyOf and copyOfRange methods do not instantiate a new object, so before calling them, you'll need to explicitly instantiate your target array.

The Arrays.copyOf method starts at index 0 in a specified source array and copies up to a specified number of elements to a returned destination array. The copied elements have the same indices in both arrays. The specified number of elements also establishes the destination array's length. If the destination array's length exceeds the source array’s length, the extra elements in the destination array get default values. Here’s how you could use it to copy Figure 9.5’s pricesJanuary array to the pricesFebruary array:

```
System.arraycopy(pricesJanuary, 0, pricesFebruary, 0, 6);
```

The first argument is the source array’s name. The second argument is the destination array’s length. Overloaded versions of this method permit elements to be any type of primitive or object, and the destination array’s elements automatically become the same type as the source array’s elements.

The Arrays.copyOfRange method copies elements in a specified range from a specified source array to a returned destination array. Here’s how you could use it to copy Figure 9.5’s pricesJanuary array to the pricesFebruary array:

```
double[] pricesFebruary = Arrays.copyOfRange(
  pricesJanuary, 0, 6);
```

As before, the first argument is the source array name. The second argument is the index of the first element to be copied from the source array. The third argument is the index of the last element to be copied from the source array, plus 1. This makes the total number of elements copied equal to the third argument minus the second argument. This difference determines the length of the destination array. If the length of the destination array exceeds the length of the source array, the extra elements in the generated destination array get default values. Overloaded versions of this method permit elements to be any type of primitive or object, and the destination array's elements automatically become the same type as the source array's elements.

9.6 Problem Solving with

Array Case Studies

Learn by example.

In this section, we present two array-based case studies. For each case study, we present a problem and then examine its solution. The point of these case studies isn't so much that you memorize the details. The point is that you get a feel for how to solve array-oriented problems. Then when you're a programmer in the real world, you'll have a "bag of tricks" that you can draw from. You'll probably have to modify the case-study solutions to make them fit your specific real-world problems, but that's OK. You've got to earn your keep, after all.

Shifting Array-Element Values

Consider the hours array in Figure 9.6. The hours array contains the scheduled work hours for a person for a 31-day period. The first element (hours[0]) contains the scheduled work hours for the person for the current day. The last element (hours[30]) contains the scheduled work hours for the person for the day that's 30 days in the future. At the beginning of each new day, the work hours need to shift to lower-index positions. For example, the hours[1] value needs to shift to the hours[0] element. That should make sense when you realize that when you're going to a new day, you need to make what was the next day's scheduled hours, hours[1], become the current day's scheduled hours, hours[0].

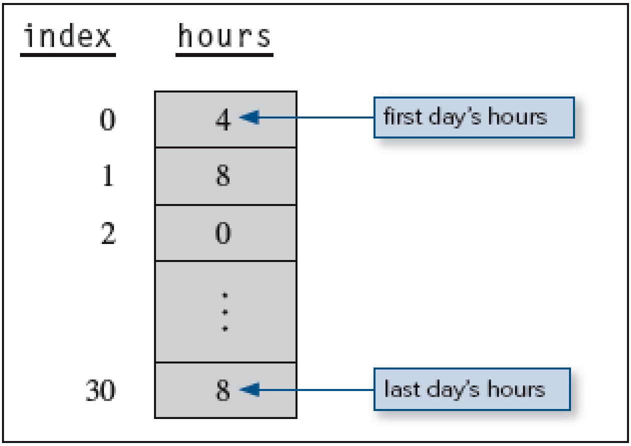
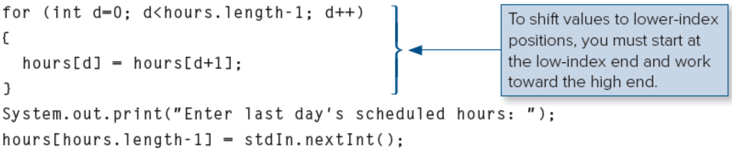


Figure 9.6 Array of scheduled work hours for the next 31 days

Now let's look at Java code that performs this shifting operation. We want to shift each hours element value to its adjacent lower-indexed element. In other words, we want to copy the second element's value into the first element, copy the third element's value into the second element, and so on. Then we want to assign a user-entered value to the last element. Here's the code:



```
for (int d=0; d<hours.length-1; d++)
{
  hours[d] = hours[d+1];
}
System.out.print("Enter last day's scheduled hours: ");
hours[hours.length-1] = stdIn.nextInt();
```

To shift values to lower-index positions, you must start at the low-index end and work toward the high end.

There are several things to note about this code fragment. It’s OK to use an expression inside the [ ]’s—we use hours[d+1] to access the element after the hours[d] element. Notice how we shift elements at the low-index end first. What would happen if you started the shifting at the high-index end? You’d overwrite the next element you wanted to move and end up filling the entire array with the value that was originally in the highest element. Not good.

Calculating a Moving Average

Borrow code and modify it.

Let’s now borrow code from the above example and apply it to another problem. Suppose you need to present a four-day moving average of the Dow Jones Industrial Average (DJIA) at the end of each business day. Assume you already have a four-element array holding the values of the DJIA at the end of the day on each of the past four days, with four-days-ago's value at index 0, three-days-ago's value at index 1, two-days-ago's value at index 2, and yesterday's value at index 3. For today's four-day moving average, you'll want the sum of the values for the last three days, plus the value for today. This means you'll need to shift everything in the array to lower-index positions and insert today's value at the high-index end. Then you'll need to sum up everything in the array and divide by the length of the array. Presumably, you'll save the shifted array somewhere and then do the same thing again at the end of each day in the future. You could do the shifting and summing in separate loops, but it's easier to do both in the same loop, as shown in Figure 9.7.

To allow for different lengths of

time, do not hardcode the array length. Instead, always use *array-name*.length. Think carefully about each boundary. Notice that the index [d+1] on the right side of the first statement in the inside for loop is 1 greater than the count variable value d. Remember that the highest index value in an array is always 1 less than the array's length. So the highest value of the count variable should be the array's length minus 2. That's why the loop-continuation condition is d<days.length-1. Also, notice that we insert the new final value for the array after the loop terminates, and then we include this final value in the sum before computing the average. Here's an example of what the program does:

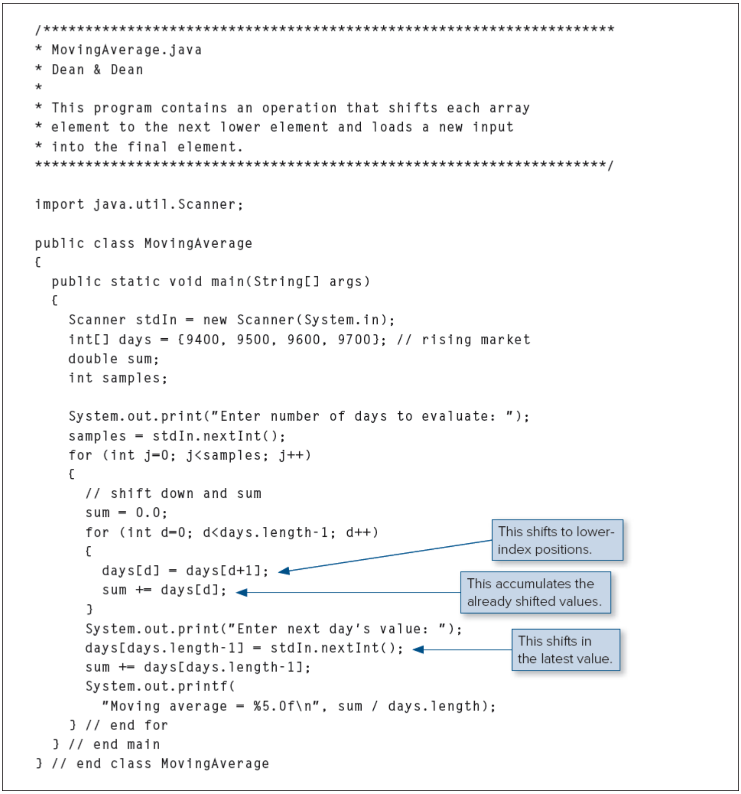
Sample session:

```
Enter number of days to evaluate: 4
Enter next day's value: 9800
Moving average = 9650
Enter next day's value: 9800
Moving average = 9725
Enter next day's value: 9700
Moving average = 9750
Enter next day's value: 9600
Moving average = 9725
```

A moving average is smoother than an instantaneous plot, but notice that its values lag behind.

There's a simpler way to do shifting. Do you remember the API arraycopy method mentioned in the previous section? You can use it to implement shifts to lower-index positions with this code fragment:

```
System.arraycopy(days, 1, days, 0, days.length-1);
System.out.print("Enter next day's value: ");
days[days.length-1] = stdIn.nextInt();
```



```
/*********************************************************************
* MovingAverage.java
* Dean & Dean
*
* This program contains an operation that shifts each array
* element to the next lower element and loads a new input
* into the final element.
*********************************************************************/

import java.util.Scanner;

public class MovingAverage
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int[] days = {9400, 9500, 9600, 9700}; // rising market
    double sum;
    int samples;

    System.out.print("Enter number of days to evaluate: ");
    samples = stdIn.nextInt();
    for (int j=0; j<samples; j++)
    {
      // shift down and sum
      sum = 0.0;
      for (int d=0; d<days.length-1; d++)
      {
        days[d] = days[d+1];
        sum += days[d];
      }
      System.out.print("Enter next day's value: ");
      days[days.length-1] = stdIn.nextInt();
      sum += days[days.length-1];
      System.out.printf(
        "Moving average = %5.0f\n", sum / days.length);
    } // end for
  } // end main
} // end class MovingAverage
```

This shifts to lower-index positions.

This accumulates the already shifted values.

This shifts in the latest value.

Figure 9.7 Calculation of a moving average

Conceptually, the arraycopy method copies everything from element 1 to the last element into a temporary array, and then copies it from this temporary array back into the original array starting at element 0. This eliminates the inner for loop in Figure

9.7. Unfortunately, we also used the inner for loop to compute the sum needed for the average. But there's a trick you can use, and it makes a program like this more efficient when the array is very large. If you keep track of the sum of all the elements in the array, each time you shift the array element values, you can just correct the sum, rather than completely recomputing it. To correct the sum, subtract the value shifted out and add the value shifted in, like this:

```
sum -= days[0];
System.arraycopy(days, 1, days, 0, days.length-1);
System.out.print("Enter next day's value: ");
days[days.length-1] = stdIn.nextInt();
sum += days[days.length-1];
```

Histograms

In this subsection, we'll use an array as part of a histogram program. But before we present the program, a histogram overview is in order. A *histogram* is a graph that displays quantities for a set of categories. Typically, it indicates category quantities with bars—shorter bars represent smaller quantities, longer bars represent larger quantities. For example, Figure 9.8's histogram shows infant mortality rates in developed countries. Histograms are a popular way to present statistical data because they provide a quick and clear representation of the data's distribution.

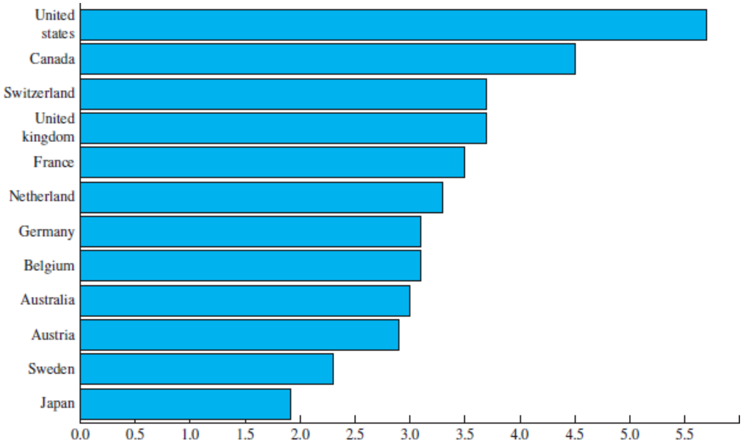


Figure 9.8 Example histogram from the World Bank and the Kaiser Family Foundation Comparison of infant mortality rate (a widely used indicator

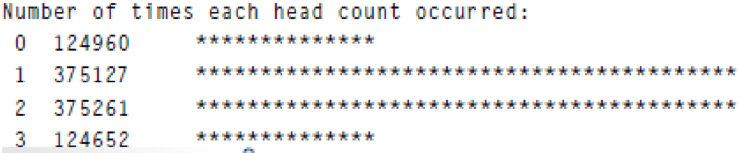
of population health) in countries with above median GDP and above median GDP per capita in at least one of the past 10 years.

Suppose you have three coins. When you flip all three, you’re curious about the probabilities of getting zero heads, one head, two heads, and three heads. In other words, you’re curious about the frequency distribution for the number of heads.

Approximate a mathematical solution with simulation.

You could calculate the frequency distribution mathematically (with the binomial distribution formula), but, instead, you decide to write a program to simulate the coin flips. If you simulate enough coin flips, then the results will approximate the mathematically calculated result.

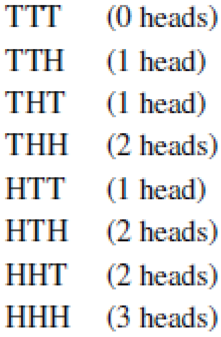
In your program, you should simulate flipping the three coins a million times. You should print the simulation results in the form of a histogram. For each of the four cases (zero heads, one head, two heads, three heads), print a series of *'s where the number of these asterisks is proportional to the number of times the case occurred. Each series of asterisks represents a histogram bar. That should make more sense by looking at this sample output:



```
Number of times each head count occurred:
 0  124960     ***************
 1  375127     *********************************************
 2  375261     *********************************************
 3  124652     ***************
```

Compare program results with predicted results.

Note the first row of asterisks. That's a horizontal "bar" that pictorially describes the number of times the zero-heads case occurred. The zero at the left is the label for the zero-heads case. The 124960 is the specific number of times the zero-heads case occurred. Or, said another way, 124960 is the *frequency* of the zero-heads case. Note that the zero-heads and three-heads frequencies (124960 and 124652, respectively) are nearly the same, and the one-head and two-heads frequencies (375127 and 375261, respectively) are also nearly the same. Also note that the zero-heads and three-heads frequencies are each approximately one third of the one-head and two-heads frequencies. It's always a good idea to use some kind of independent calculation to predict what a computer's answer should be like. For this simple problem, it's relatively easy to compute an exact answer. Assuming that "T" means "tails" and "H" means "heads," here are all the possible flipping results:



TTT	(0 heads)
TTH	(1 head)
THT	(1 head)
THH	(2 heads)
HTT	(1 head)
HTH	(2 heads)
HHT	(2 heads)
HHH	(3 heads)

Note that there is only one way to obtain zero heads and only one way to obtain three heads, but there are three ways to obtain one head and three ways to obtain two heads. So the zero-head and three-head frequencies should each be one third of the one-head and two-head frequencies. If you look at the numbers and bar lengths in the above sample output, you'll see that the computer result approximately conforms to this expectation.

See Figure 9.9's CoinFlips program. It does what we want. It simulates flipping three coins a million times, and it prints the simulation results in the form of a histogram. It uses a four-element frequency array to keep track of the number of times each head-count value occurs. Each element in the frequency array is called a *bin*. In general, a bin contains the number of occurrences of an event. For the CoinFlips program, the frequency[0] element is the first bin, and it holds the number of times none of the three coins lands heads up. The frequency[1] element is the second bin, and it holds the number of times one of the three coins lands heads up. After each three-coin-flip simulation iteration, the program adds 1 to the appropriate bin. For example, if a particular iteration generates one head, the program increments the frequency[1] bin. And if a particular iteration generates two heads, the program increments the frequency[2] bin.

```
/*********************************************************************
* CoinFlips.java
* Dean & Dean
*
* This generates a histogram of coin flips.
*********************************************************************/

public class CoinFlips
{
  public static void main(String[] args)
  {
    final int NUM_OF_COINS = 3;       // number of coins
    final int NUM_OF_REPS = 1000000;  // repetitions

    // The frequency array holds the number of times
    // a particular number of heads occurred.
    int[] frequency = new int[NUM_OF_COINS + 1];
    int heads;              // heads in current group of flips
    double fractionOfReps;  // head count / repetitions
    int numOfAsterisks;     // asterisks in one histogram bar

    for (int rep=0; rep<NUM_OF_REPS; rep++)
    {
      // perform a group of flips
      heads = 0;
      for (int i=0; i<NUM_OF_COINS; i++)
      {
        heads += (int) (Math.random() * 2);
      }
      frequency[heads]++;   // update appropriate bin
    } // end for
    System.out.println(
      "Number of times each head count occurred:");
    for (heads=0; heads<=NUM_OF_COINS; heads++)
    {
      System.out.print(
        " " + heads + "  " + frequency[heads] + " ");
      fractionOfReps = (float) frequency[heads] / NUM_OF_REPS;
      numOfAsterisks = (int) Math.round(fractionOfReps * 100);

      for (int i=0; i<numOfAsterisks; i++)
      {
        System.out.print("*");
      }
      System.out.println();
    } // end for
  } // end main
} // end class CoinFlips
```

This loop fills up the frequency bins. Each iteration simulates one group of three coin flips.

This loop prints the histogram. Each iteration prints one histogram bar.

Figure 9.9 CoinFlips program that generates a histogram for a coin-flips simulation

Let's now examine how the CoinFlips program prints the histogram asterisk bars. As specified by the second callout in Figure 9.9, the second large for loop prints the histogram. Each iteration of the for loop prints the bin label (0, 1, 2, or 3) and then the frequency for that bin. Then it computes the number of asterisks to print by dividing the frequency in the current bin by the total number of repetitions and multiplying by 100. Then it uses an inner for loop to display the computed number of asterisks.

9.7 Searching an Array

In order to use an array, you need to access its individual elements. If you know the location of the element you're interested in, then you simply access the element by putting the element's index inside square brackets. But if you don't know the location of the element, you need to search for it. For example, suppose you're writing a program that keeps track of student enrollments for the courses at your school. The program should be able to add a student, remove a student, view a student's data, and so on. All of those operations require that you first search for the student within a student ids array (even the add-a-student operation requires a search, to ensure that the student isn't already in the array). In this section, we present two techniques for searching an array.

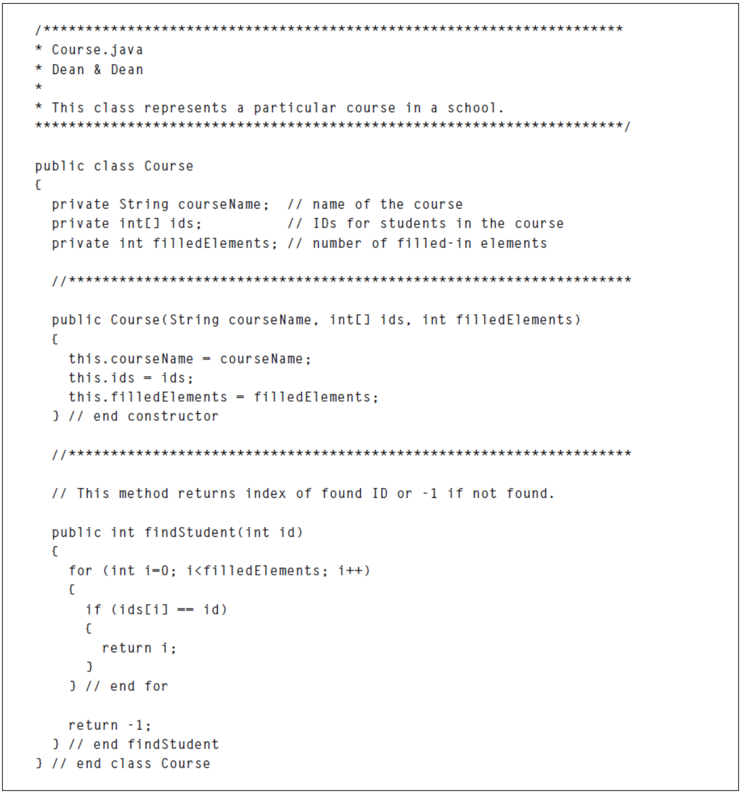
Sequential Search

If the array is short (has less than about 20 items), the best way to search it is the simplest way: Step through the array sequentially and compare the value at each array element with the searched-for value. When you find a match, do something and return. Here's a pseudocode description of the sequential-search algorithm:

```
i ← 0
while i < number of filled elements
{
  if list[i] equals the searched-for value
    <do something and stop the loop>
  increment i
}
```

Adapt generic algorithms to specific situations.

Typically, algorithms are more generic than Java implementations. Part of problem solving is the process of adapting generic algorithms to specific situations. In this case, the “do something” code will be different for different cases. The findStudent method in Figure 9.10 illustrates one implementation of the sequential-search algorithm. This particular method might be part of a Course class that implements an academic course. The Course class stores a course’s name, an array of student IDs for the students enrolled in the course, and the number of students in the course. The findStudent method searches for a given student ID within the student ids array. If it finds the student ID, it returns its index. Otherwise, it returns −1. Note how findStudent's code matches the sequential-search algorithm's logic. In particular, note how findStudent implements <*do something and stop the loop*> with a return i statement. The return i implements "do something" by returning the index of the found student ID. It implements "stop the loop" by returning from the method and terminating the loop simultaneously.



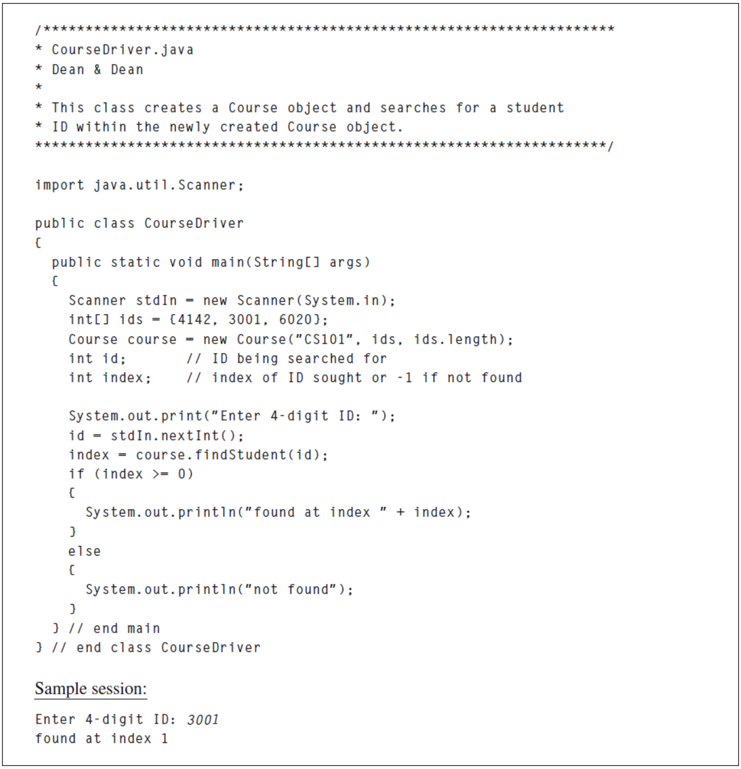
```
/*********************************************************************
* Course.java
* Dean & Dean
*
* This class represents a particular course in a school.
*********************************************************************/

public class Course
{
  private String courseName;  // name of the course
  private int[] ids;          // IDs for students in the course
  private int filledElements; // number of filled-in elements

  //******************************************************************

  public Course(String courseName, int[] ids, int filledElements)
  {
    this.courseName = courseName;
    this.ids = ids;
    this.filledElements = filledElements;
  } // end constructor

  //******************************************************************

  // This method returns index of found ID or -1 if not found.

  public int findStudent(int id)
  {
    for (int i=0; i<filledElements; i++)
    {
      if (ids[i] == id)
      {
        return i;
      }
    } // end for

    return -1;
  } // end findStudent
} // end class Course
```

Figure 9.10 Course class, which includes a sequential search method (findStudent)

In examining the findStudent method, you might be asking yourself “What is the practical use for the returned index?” To do anything with an ID in the ids array (add an ID, remove an ID, etc.) you need to know the ID’s index. If you don’t know the

ID's index in advance, the findStudent method finds the ID's index for you. Are you still asking yourself "What is the practical use for the returned −1 when the ID is not found?" The −1 can be used by the calling module to check for the case of an invalid student ID.

Figure 9.11's CourseDriver class drives Figure 9.10's Course class. The CourseDriver class is fairly straightforward. It creates an array of student IDs, stores the array in a Course object, prompts the user for a particular student ID, and then calls findStudent to see whether that particular student is taking the course. To keep things simple, we use an initializer to create the ids array. For a more general-purpose driver, you might want to replace the initializer with a loop that repeatedly prompts the user to enter a student ID or −1 to quit. If you choose that option, then you'd need to store the number of filled elements in a filledElements variable and pass the filledElements variable as the third argument in the Course constructor call. This is what the constructor call would look like:



```
/*************************************************************************
* CourseDriver.java
* Dean & Dean
*
* This class creates a Course object and searches for a student
* ID within the newly created Course object.
*************************************************************************/

import java.util.Scanner;

public class CourseDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int[] ids = {4142, 3001, 6020};
    Course course = new Course(”CS101”, ids, ids.length);
    int id;       // ID being searched for
    int index;    // index of ID sought or -1 if not found

    System.out.print(”Enter 4-digit ID: ”);
    id = stdIn.nextInt();
    index = course.findStudent(id);
    if (index >= 0)
    {
      System.out.println(”found at index ” + index);
    }
    else
    {
      System.out.println(”not found”);
    }
  } // end main
} // end class CourseDriver
```

Sample session:

```
Enter 4-digit ID: 3001
found at index 1
```

Figure 9.11 Driver for Course program, which illustrates a sequential search

Course course = new

Course("CS101", ids, filledElements);

Binary Search

Page 406

If you have an array with a large number of array elements, like 100,000, a sequential search typically takes quite a long time. If such an array has to be searched many times, it's often worthwhile to use a binary search. *Binary search* gets its name from the way that it bisects a list of values and narrows its search to just half of the bisected list.

For a binary search to work on an array, the array must be sorted so that everything is in some kind of alphabetical or numerical order. The next section describes one of the many available sorting methods. This initial sorting takes more time than a single sequential search, but you have to do it only once.

After the array has been sorted,

you can use a binary search to find values in the array very quickly—even when the array is extremely long. The worst (slowest) case is when the searched-for value is not in the array. A worst-case sequential search takes an amount of time proportional to the array length. In comparison, a worst-case binary search takes an amount of time proportional to the logarithm of the array length. When an array is very long, the difference between linear and logarithmic is huge. For example, suppose the length is 100,000. It works out that log2(100,000) ≈ 17. Because 17 is about 6,000 times smaller than 100,000, binary search is approximately 6,000 times faster than sequential search for a 100,000-element array.

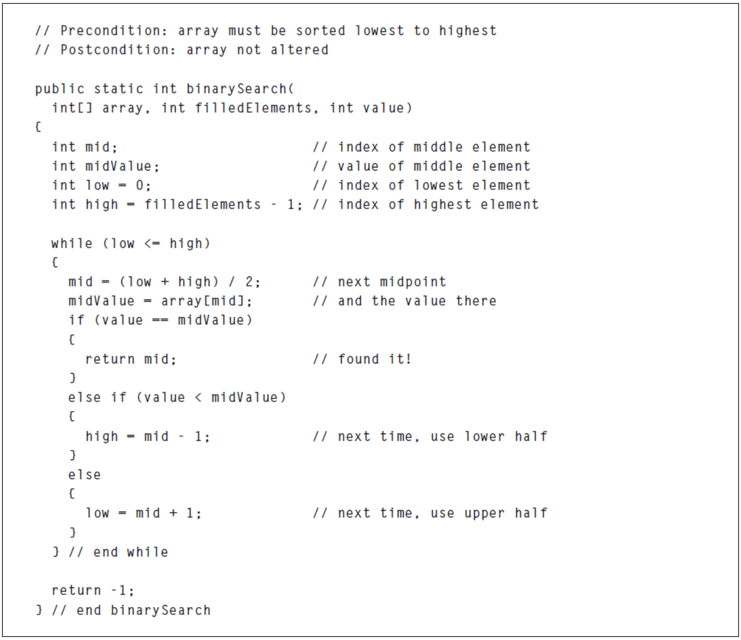
Note the binarySearch method in Figure 9.12, and, in particular, note its static modifier. You can use either an instance method or a static method to implement searching. In the previous subsection, we implemented searching with an instance method. This time, we implement searching with a static method, which is appropriate if you want a method to be used generically. To make it generic (that is, to make it usable by different programs), you should put the method in a separate class and make the method a static method. Because it's a static method, different programs can call the binarySearch method easily, using binarySearch's class name, rather than using a calling object. For example, if you put the binarySearch method in a Utilities class, you would call binarySearch like this:

```
Utilities.binarySearch(
  array-name, number-of-filled-elements, searched-for-value);
```

In the binarySearch method call, note the array argument. Being a static method, binarySearch cannot access instance variables. More specifically, it cannot access the searched array as an instance variable. So the searched array must be passed in as an argument. This allows the method to be used for arrays defined outside of the method's class. But as you may recall from Section 7.5, passing a reference (an array name is a reference) allows mutations to propagate back to the caller's code. Not to worry—the postcondition comment at the top of Figure 9.12 guarantees that this particular method will not alter its array argument.

Split up a problem into smaller problems.

The binary search algorithm is based on an ancient strategy—*divide and conquer*. You first identify the middle element in the sorted array. You then figure out whether the searched-for value belongs before or after the middle element. If it belongs before the middle element, you narrow the search range to the lower half of the array (the half with the smaller-indexed elements). If, on the other hand, the searched-for value belongs after the middle element, you narrow the search range to the upper half of the array. You then repeat the process. In other words, within the narrowed-down half of the array, you identify the middle element, figure out whether the searched-for value belongs before or after the middle element, and narrow the search range accordingly. Every time you do this, you cut the problem in half, and this enables you to zero in quickly on the searched-for value—if it's there at all. Splitting the array in half is the "divide" part of "divide and conquer." Finding the searched-for value within one of the halves is the "conquer" part.



```
// Precondition: array must be sorted lowest to highest
// Postcondition: array not altered

public static int binarySearch(
  int[] array, int filledElements, int value)
{
  int mid;                        // index of middle element
  int midValue;                   // value of middle element
  int low = 0;                    // index of lowest element
  int high = filledElements - 1;  // index of highest element

  while (low <= high)
  {
    mid = (low + high) / 2;       // next midpoint
    midValue = array[mid];        // and the value there
    if (value == midValue)
    {
      return mid;                 // found it!
    }
    else if (value < midValue)
    {
      high = mid - 1;             // next time, use lower half
    }
    else
    {
      low = mid + 1;              // next time, use upper half
    }
  } // end while

  return -1;
} // end binarySearch
```

Figure 9.12 Method that performs a binary search of an array already sorted in ascending order

Now let’s see how the binarySearch method implements the divide-and-conquer algorithm. The method declares mid, low, and high variables that keep track of the indexes for the middle element and the two elements at the ends of the array’s search range. For an example, see the left drawing in Figure 9.13. Using a while loop, the method repeatedly calculates mid (the index of the middle element) and checks whether the mid element's value is the searched-for value. If the mid element's value is the searched-for value, then the method returns the mid index. Otherwise, the method narrows the search range to the low half or the high half of the array. For an example of that narrowing process, see Figure 9.13. The method repeats the loop until either the searched-for value is found or the search range shrinks to the point where low's index is greater than high's index.

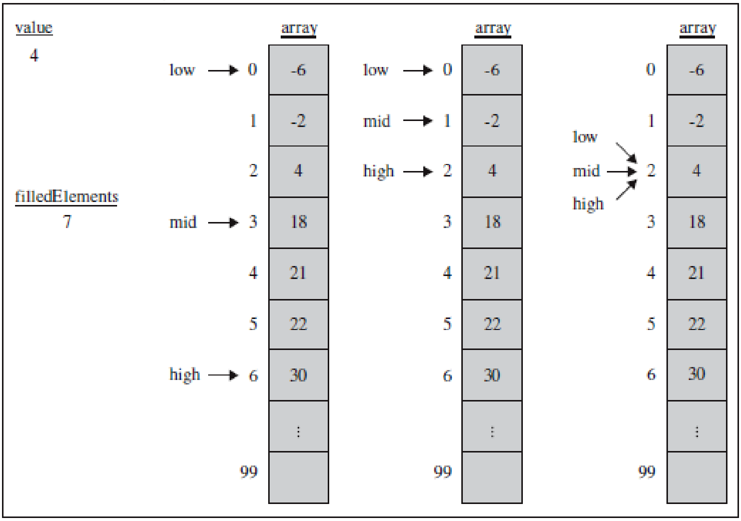

value
4
filledElements
7
array
low
mid
high
0
1
2
3
4
5
6
99
-6
-2
4
18
21
22
30
⋮

Figure 9.13 Example execution of Figure

9.12’s binarySearch method

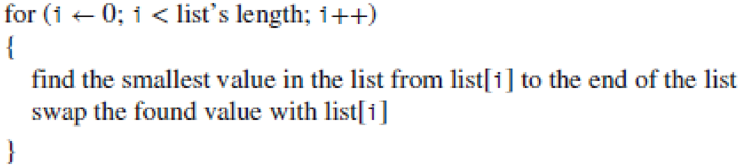
9.8 Sorting an Array

Computers are particularly good at storing large quantities of data and accessing that data quickly. As you learned in the previous section, binary search is an effective technique for finding and accessing data quickly. In order to prepare the data for binary search, the data must be sorted. Sorting data is done not only for binary search purposes. Computers also sort data so that it’s easier to display in a user-friendly fashion. If you look at the e-mails in your inbox, aren’t they normally sorted by date, with the most recent e-mail first? Most e-mail organizers allow you to sort your e-mails using other criteria as well, such as using the “from” person or using the size of the e-mail. In this section, we describe the basics of how sorting is performed. We first present a sorting algorithm, and we then present its implementation in the form of a program that sorts the values in an array.

Selection Sort

There are many different sorting algorithms with varying degrees of complexity and efficiency. Sometimes, the best way to solve a problem on a computer is the way a person would naturally solve the problem by hand. To illustrate this idea, let's examine how people tend to sort cards when playing a card game. Assume you want to sort in ascending order with the smallest cards going at the left. You search for and select the smallest card and move it to the leftmost position. The left side is where you keep the cards that have been sorted already. You then search for the next smallest card, but in so doing, you look only at cards that are in the unsorted portion of the card group. You move the found card to the second position on the left side of the card group. You repeat this search-and-move process until there are no more cards in the unsorted portion of the card group. What we’ve described is an overview of the *selection sort* algorithm.

As a first step in implementing the selection sort logic, let’s examine a pseudocode solution. Above, we said to “repeat the search-and-move process.” Whenever there’s a repetition, you should think about using a loop. The following algorithm uses a loop for repeating the search-and-move process. Note how the index i keeps track of where the search starts. The first time through the loop, the search starts at the first element (at index 0). The next time, the search starts at the second position. Each time through the loop, you find the smallest value and move it to the sorted portion of the list (the i also tells you where the smallest value should go).



```
for (i ← 0; i < list's length; i++)
{
  find the smallest value in the list from list[i] to the end of the list
  swap the found value with list[i]
}
```

The loop's last line says "swap the found value with list[i]." That takes care of moving the smallest value to the sorted portion of the list. The swapping mechanism is probably a departure from what most people do when sorting cards. Most people would insert the just-found smallest value and shift the other cards to make room for it. Instead, we swap instead of shift because we want to keep things simple, and that's how the simple selection-sort algorithm works. Inserting and shifting mimics a different sorting algorithm, *insertion sort*, which is presented in one of the exercises at the end of the chapter.

In the prior pseudocode, there's an implied loop within the given for loop. The "find the smallest value" operation requires looping through the list and checking each element to see if it's the smallest value. The length of that inner loop shrinks as the outer loop progresses. A picture is worth a thousand words, so we provide Figure 9.14 to show the selection sort algorithm in action. The five pictures show the different stages of a list being sorted using the selection sort algorithm. The white portion of each stage is unsorted. The original stage on the left is all white, indicating that it is entirely unsorted. The shaded portion of each stage is sorted. The stage on the right is all shaded, indicating that it is entirely sorted. The bidirectional arrows show what happens after a smallest value is found. The smallest value (at the bottom of the bidirectional arrow) gets swapped up to the top of the unsorted portion of the list. For example, in going from the first picture to the second picture, the smallest value, −3, gets swapped up to 5’s position at the top of the unsorted portion of the list.

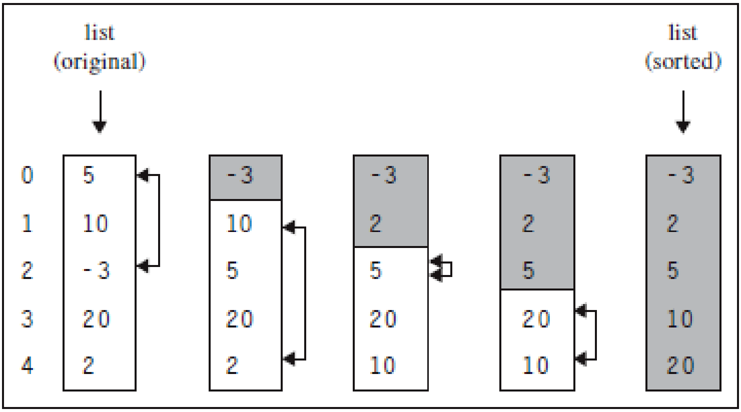

list
(original)
list
(sorted)
0
1
2
3
4
5
10
-3
20
2
-3
10
5
20
2
-3
2
5
20
10
-3
2
5
20
10
-3
2
5
10
20

Figure 9.14 Stages in an example execution of the selection sort algorithm

Now let’s implement a Java version of the selection sort algorithm. You can use either an instance method or a static method. In the previous section, we implemented binary search with a static method. For additional practice, we’ll do the same here for selection sort. By implementing selection sort with a static method, you can easily call it from any program that needs to sort a list of numbers—just prefix the method call with class name dot.

See the Sort class in Figure 9.15. Note how the sort method body mimics the pseudocode very closely because the sort method uses top-down design. Rather than include the search-for-the-smallest-value code within the sort method, the sort method calls the indexOfNextSmallest helper method. Rather than include the element-swapping code within the sort method, the sort method calls the swap helper method. The only substantive difference between the sort method and the sort algorithm is that the sort method's for loop stops iterating one element before the bottom of the array. That's because there's no need to perform a search when you're at the last element (you already know that the last element is the minimum value for the remainder of the list). We didn't worry about such efficiency details with the algorithm because algorithms are more about basic logic rather than off-by-one details.

Passing Arrays as Arguments

Figure 9.16 contains a driver for Figure 9.15's Sort class. Most of the code is fairly straightforward, but please take note of the studentIds argument in the Sort.sort method call. This is another example of passing an array to a method. An array is an object, and as such, studentIds is a reference to an array object. As indicated in Section 7.5, a reference argument (in a method call) and its corresponding reference parameter (in a method heading) point to the same object. So if you update the reference parameter's object from within the method, you simultaneously update the reference argument's object in the calling module. Applying that thinking to the Sort program, when you pass the studentIds reference to the sort method and sort the array there, there's no need to return the updated (sorted) array with a return statement. That's because the studentIds reference points to the same array object that is sorted within the sort method. Thus, we do not include a return statement in the sort method, and the method works just fine.

Sorting with a Java API Method

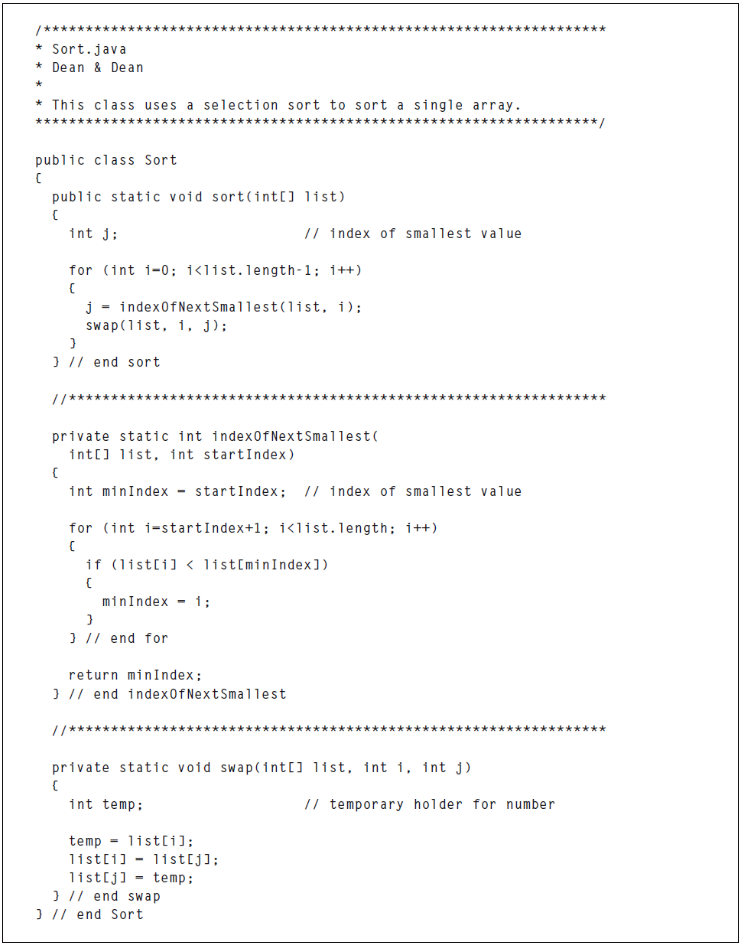
Check for efficient API methods.

When an array has more than about 20 elements, it's better to use an algorithm that's more efficient than the relatively simple selection sort algorithm just described. And sure enough, the Java API has a sorting method that uses a more efficient sorting algorithm. It’s the sort method in the Arrays class.

Here’s skeleton code for how you might use the Arrays class’s sort method:

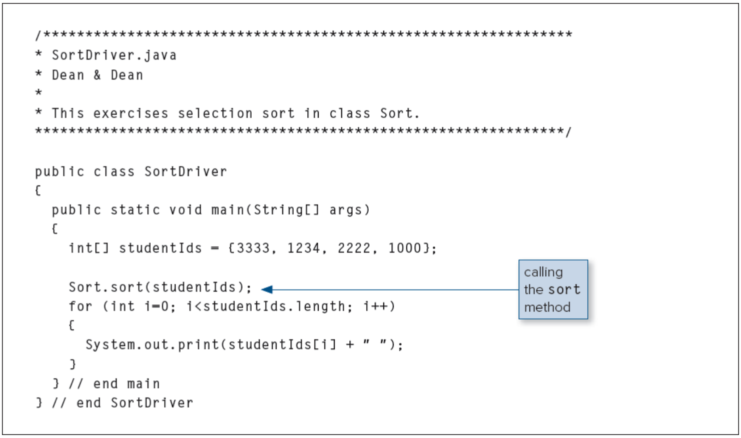
```
import java.util.Arrays;
...
  int[] studentIds = {...};
  ...
  Arrays.sort(studentIds);
```

We recommend that you use this API method for heavy-duty sorting. It’s an overloaded method, so it also works for arrays of other types of primitive variables. Chapter 11 describes another sorting technique that is almost as efficient as the Java API sorting method, and it’s relatively easy to understand. That chapter also describes an alternate implementation of binary search. At this point, if you wish, you can safely jump there.



```
/*********************************************************************
* Sort.java
* Dean & Dean
*
* This class uses a selection sort to sort a single array.
*********************************************************************/

public class Sort
{
  public static void sort(int[] list)
  {
    int j;                      // index of smallest value

    for (int i=0; i<list.length-1; i++)
    {
      j = indexOfNextSmallest(list, i);
      swap(list, i, j);
    }
  } // end sort

  //*****************************************************************

  private static int indexOfNextSmallest(
    int[] list, int startIndex)
  {
    int minIndex = startIndex;  // index of smallest value

    for (int i=startIndex+1; i<list.length; i++)
    {
      if (list[i] < list[minIndex])
      {
        minIndex = i;
      }
    } // end for

    return minIndex;
  } // end indexOfNextSmallest

  //*****************************************************************

  private static void swap(int[] list, int i, int j)
  {
    int temp;                   // temporary holder for number

    temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  } // end swap
} // end Sort
```

Figure 9.15 Sort class containing a method that sorts an array of integers in ascending order



```
/***************************************************************
* SortDriver.java
* Dean & Dean
*
* This exercises selection sort in class Sort.
***************************************************************/

public class SortDriver
{
  public static void main(String[] args)
  {
    int[] studentIds = {3333, 1234, 2222, 1000};

    Sort.sort(studentIds);
    for (int i=0; i<studentIds.length; i++)
    {
      System.out.print(studentIds[i] + " ");
    }
  } // end main
} // end SortDriver
```

calling the `sort` method

Figure 9.16 Driver that exercises the sort method in Figure 9.15

Page 412

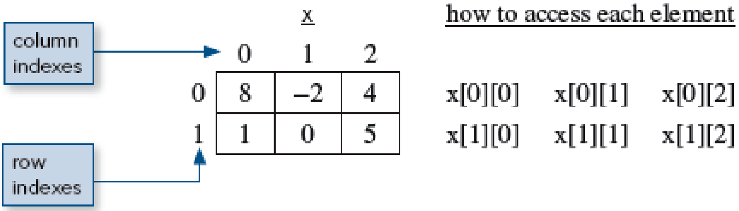
9.9 Two-Dimensional Arrays

Arrays are good for grouping related data together. Up to this point, we've grouped the data together using standard one-dimensional arrays. If the related data is organized in a table format, consider using a two-dimensional array. This section describes two-dimensional arrays and higher-dimensional arrays.

Two-Dimensional Array Syntax

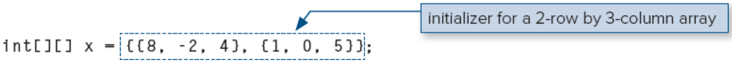
Two-dimensional arrays use the same basic syntax as one-dimensional arrays except for a second pair of square brackets ([]). Each pair of square brackets contains one index. According to standard programming practice, the first index identifies the row and the second index identifies the column position within a row.

For example, here's a two-row by three-column array named x:


column indexes
row indexes
x
0 1 2
0 8 –2 4
1 1 0 5
how to access each element
x[0][0] x[0][1] x[0][2]
x[1][0] x[1][1] x[1][2]

The items at the right, under the "how to access" column heading, show how to access each of the six elements in the array. So to access the value 5, at row index 1 and column index 2, you specify x[1][2].

As with one-dimensional arrays, there are two ways to assign values into a two-dimensional array's elements. You can use an array initializer, where the element assignment is part of the array's declaration. Or you can use standard assignment statements, where the assignment statements are separate from the array's declaration and creation. We'll describe the array initializer technique first. Here's how you can declare the above two-dimensional x array and assign values into its elements, using an array initializer:



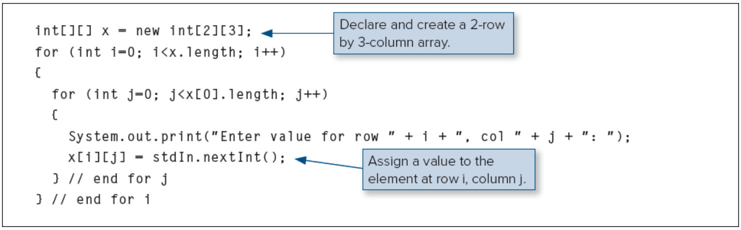
```
int[][] x = {{8, -2, 4}, {1, 0, 5}};
```

initializer for a 2-row by 3-column array

Note that the array initializer contains two inner groups, where each inner group represents one row. {8, -2, 4} represents the first row. {1, 0, 5} represents the second row. Note that elements and groups are separated with commas, and each inner group and the entire set of inner groups are surrounded by braces.

You can use the array initializer technique only if you know the assigned values when you first declare the array. Otherwise, you need to provide array element assignment statements that are separate from the -array's declaration and creation. For example, Figure 9.17's code fragment declares and creates the x array in one statement, and assigns values to x's elements in a separate statement, inside nested for loops.

When working with two-dimensional arrays, it's very common to use nested for loops. In Figure 9.17, note the outer for loop with index variable i and the inner for loop with index variable j. The outer for loop iterates through each row, and the inner for loop iterates through each element within a particular row.



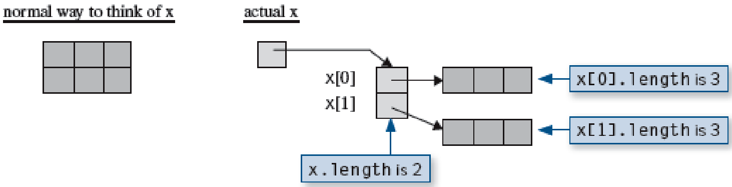
```
int[][] x = new int[2][3];
for (int i=0; i<x.length; i++)
{
  for (int j=0; j<x[0].length; j++)
  {
    System.out.print("Enter value for row " + i + ", col " + j + ": ");
    x[i][j] = stdIn.nextInt();
  } // end for j
} // end for i
```

Declare and create a 2-row by 3-column array.

Assign a value to the element at row i, column j.

Figure 9.17 Assigning values to a two-dimensional array using nested for loops and the length property

Figure 9.17's first line declares x to be a 2-row by 3-column array with 6 total elements. So you might expect the first for loop's x.length property to hold a 6. Not so. Even though it's normal (and useful) to think of x as a rectangular box that holds 6 int elements, x is actually a reference to a 2-element array, and each of the two elements is a reference to its own 3-element array of ints. This picture illustrates what we're talking about:

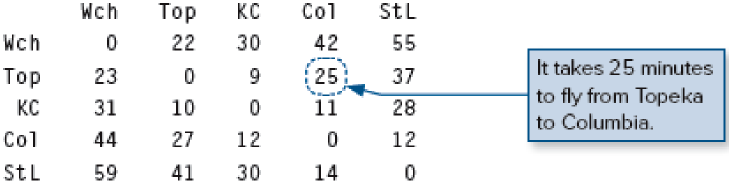


Because x is actually a reference to a

2-element array, x.length holds the value 2. Or thinking about x in the “normal” way (above left picture), x.length holds the number of rows in x. As you can see above, x[0] is a reference to a 3-element array. Thus, x[0].length holds the value 3. Or, thinking about x in the “normal” way (above left picture), x[0].length holds the number of columns in x. The point of all this is that the length property can be used for iterating through the elements in a two-dimensional array. In Figure 9.17, note how the first loop uses x.length to iterate through the rows in x, and note how the second loop uses x[0].length to iterate through the columns in each row.

Example

Let’s put these two-dimensional array concepts into practice by using a two-dimensional array in the context of a complete program. The program, built for a Kansas and Missouri airline company, tells customers when airplanes are expected to arrive at various Kansas and Missouri airports. It uses a two-dimensional array to store flight times between cities, and it displays output like this:



```
       Wch   Top   KC   Col   StL
Wch      0    22   30    42    55
Top     23     0    9    25    37
 KC     31    10    0    11    28
Col     44    27   12     0    12
StL     59    41   30    14     0
```

It takes 25 minutes to fly from Topeka to Columbia.

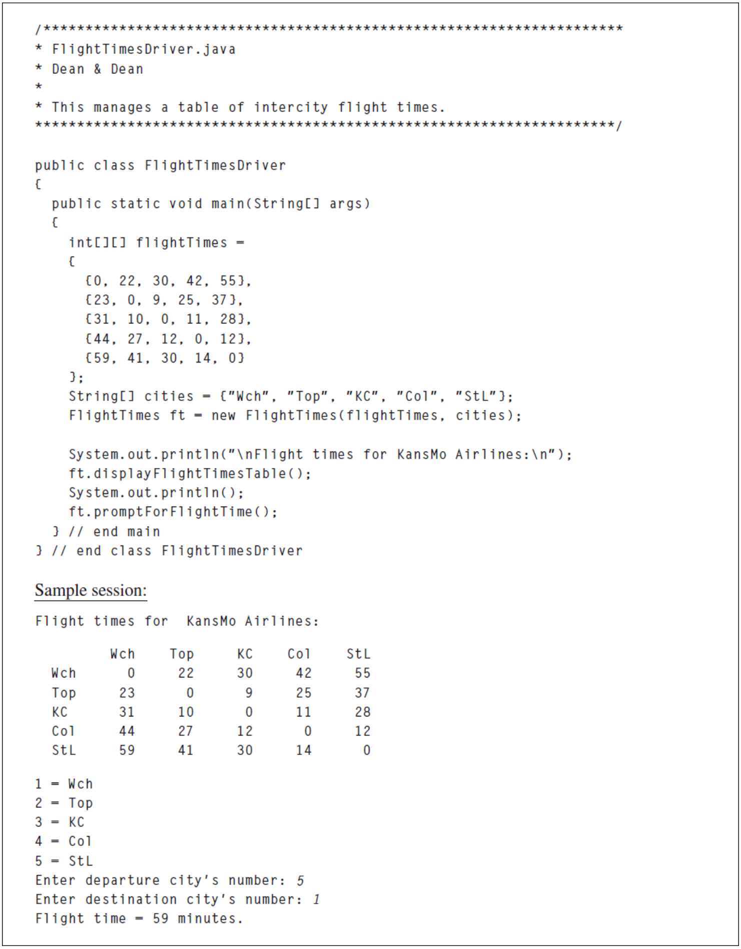
Different rows correspond to different cities of origin. Different columns correspond to different cities of destination. The labels are abbreviations for city names: “Wch” stands for Wichita, Kansas. “Top” stands for Topeka, Kansas. “KC” stands for Kansas City, Missouri. “Col” stands for Columbia, Missouri. “StL” stands for St. Louis, Missouri. Thus, for example, it takes 25 minutes to fly from Topeka to Columbia. How long does it take to go the other way, from

Columbia to Topeka? 27 minutes. Columbia to Topeka takes longer because the trip goes east to west, and airplanes have to contend with headwinds from North America's west-to-east jet stream.

Let's analyze the program by starting with Figure 9.18's FlightTimesDriver class. Note how the main method declares and creates a flightTimes table with a two-dimensional array initializer. And note how the initializer puts each table row on a line by itself. That's not required by the compiler, but it makes for elegant, self-documenting code. It is self-documenting because readers can easily identify each row of table data by looking at a single row of code. After initializing the flightTimes table, main initializes a one-dimensional array of city names and then calls the FlightTimes constructor, the displayFlightTimesTable method, and the promptForFlightTime method. We'll discuss the constructor and those two methods next.

Figures 9.19a and 9.19b contain the heart of the program—the FlightTimes class. In Figure 9.19a, the constructor initializes the flightTimes and cities instance variable arrays with the data passed to it by the driver's constructor call. Note that it assigns the passed-in ft and c array references to the instance variables using the = operator. Previously, you learned to use a for loop, not the = operator, to make a copy of an array. Why is the = operator acceptable here? Because there's no need to make a second copy of these arrays. After the constructor's first assignment operation, the flightTimes instance variable array reference and the ft parameter array reference point to the same array object. And that's appropriate. Likewise, after the constructor's second assignment operation, the cities instance variable array reference and the c parameter array reference point to the same array object.

Page 415



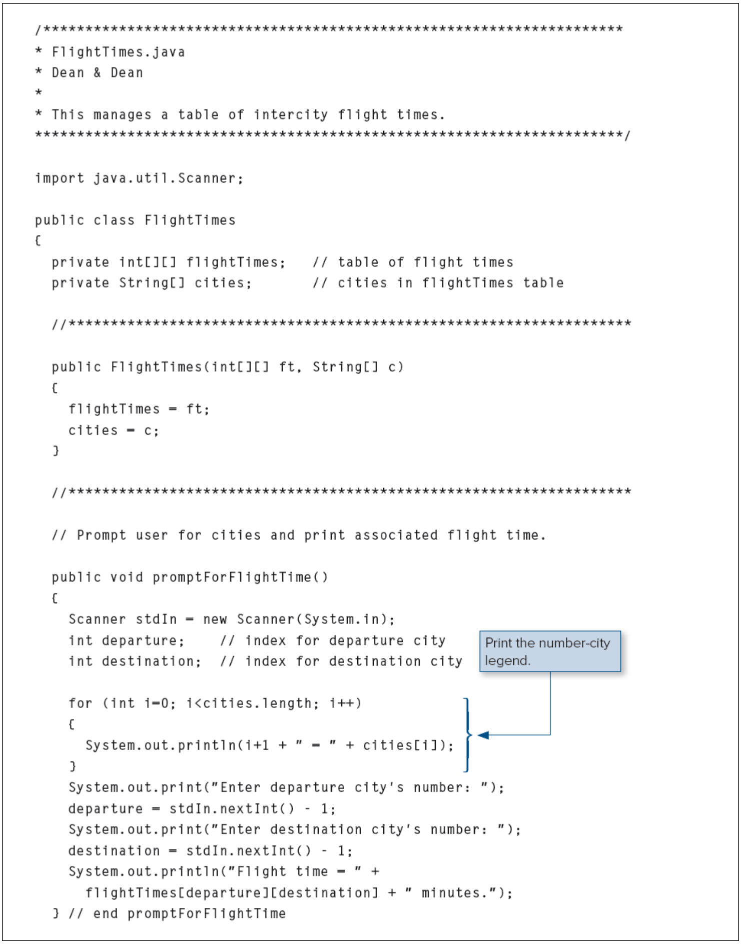
```
/***********************************************************************
* FlightTimesDriver.java
* Dean & Dean
*
* This manages a table of intercity flight times.
***********************************************************************/

public class FlightTimesDriver
{
  public static void main(String[] args)
  {
    int[][] flightTimes =
    {
      {0, 22, 30, 42, 55},
      {23, 0, 9, 25, 37},
      {31, 10, 0, 11, 28},
      {44, 27, 12, 0, 12},
      {59, 41, 30, 14, 0}
    };
    String[] cities = {"Wch", "Top", "KC", "Col", "StL"};
    FlightTimes ft = new FlightTimes(flightTimes, cities);

    System.out.println("\nFlight times for KansMo Airlines:\n");
    ft.displayFlightTimesTable();
    System.out.println();
    ft.promptForFlightTime();
  } // end main
} // end class FlightTimesDriver
```

Sample session:

```
Flight times for  KansMo Airlines:

         Wch    Top     KC    Col    StL
  Wch      0     22     30     42     55
  Top     23      0      9     25     37
  KC      31     10      0     11     28
  Col     44     27     12      0     12
  StL     59     41     30     14      0

1 = Wch
2 = Top
3 = KC
4 = Col
5 = StL
Enter departure city's number: 5
Enter destination city's number: 1
Flight time = 59 minutes.
```

Figure 9.18 Driver of FlightTimes class in Figures 9.19a and 9.19b

Page 416



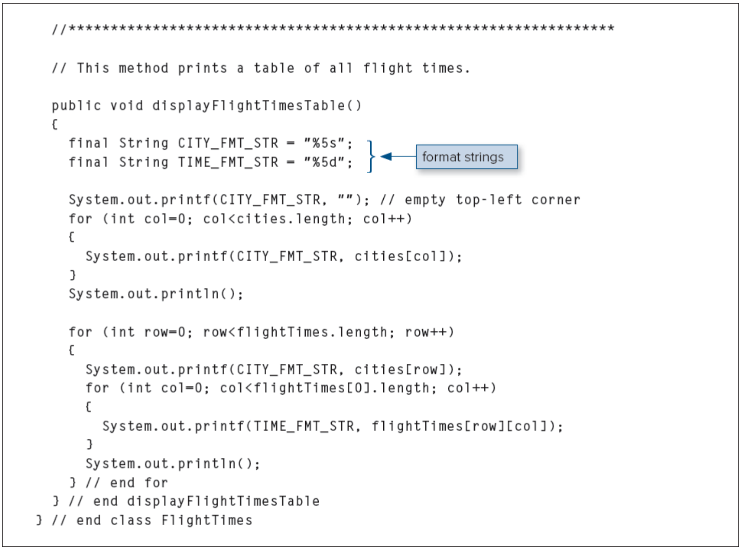
```
/*********************************************************************
* FlightTimes.java
* Dean & Dean
*
* This manages a table of intercity flight times.
*********************************************************************/

import java.util.Scanner;

public class FlightTimes
{
  private int[][] flightTimes;   // table of flight times
  private String[] cities;       // cities in flightTimes table

  //*******************************************************************

  public FlightTimes(int[][] ft, String[] c)
  {
    flightTimes = ft;
    cities = c;
  }

  //*******************************************************************

  // Prompt user for cities and print associated flight time.

  public void promptForFlightTime()
  {
    Scanner stdIn = new Scanner(System.in);
    int departure;    // index for departure city
    int destination;  // index for destination city

    for (int i=0; i<cities.length; i++)
    {
      System.out.println(i+1 + " = " + cities[i]);
    }
    System.out.print("Enter departure city's number: ");
    departure = stdIn.nextInt() - 1;
    System.out.print("Enter destination city's number: ");
    destination = stdIn.nextInt() - 1;
    System.out.println("Flight time = " +
      flightTimes[departure][destination] + " minutes.");
  } // end promptForFlightTime
```

Print the number-city legend.

Figure 9.19a FlightTimes class that displays intercity flight times—part A

Page 417

Figure 9.19a's promptForFlightTime method prompts the user for a departure city and a destination city and prints the flight time for that flight. More specifically, it prints a legend of numbers and their associated city names (1 = Wichita, 2 = Topeka, and so on), it prompts the user to enter numbers for the departure and destination cities, and it prints the flight time between the specified cities. Note how user-entered city numbers start with 1 rather than 0 (1 = Wichita). That makes the program more user-friendly because people usually prefer to start counting at one rather than zero. Internally, the program stores city names in an array. Since all arrays start with a 0 index, the program has to translate between user-entered city numbers (which start at 1) and city array indexes (which start at 0). Note how that's done with +1 and -1 in the promptForFlightTime method.



```
  //*****************************************************************

  // This method prints a table of all flight times.

  public void displayFlightTimesTable()
  {
    final String CITY_FMT_STR = "%5s";
    final String TIME_FMT_STR = "%5d";

    System.out.printf(CITY_FMT_STR, ""); // empty top-left corner
    for (int col=0; col<cities.length; col++)
    {
      System.out.printf(CITY_FMT_STR, cities[col]);
    }
    System.out.println();

    for (int row=0; row<flightTimes.length; row++)
    {
      System.out.printf(CITY_FMT_STR, cities[row]);
      for (int col=0; col<flightTimes[0].length; col++)
      {
        System.out.printf(TIME_FMT_STR, flightTimes[row][col]);
      }
      System.out.println();
    } // end for
  } // end displayFlightTimesTable
} // end class FlightTimes
```

Figure 9.19b FlightTimes class that displays intercity flight times—part B

Figure 9.19b's displayFlightTimesTable method displays the flight times table. In doing so, it employs an interesting formatting technique. First, look at the two local named constants, which are separately defined format strings. You have been using literal format strings embedded in strings of text for some time now in the arguments of printf method calls. But instead of embedding literal format strings, sometimes it's easier to understand if you declare them separately as named constants. If you go back and count the spaces in the six-column table of flight times, you'll see that each column is exactly 5 spaces wide. So the labels at the top of the columns and the numbers in the columns must both be formatted to use exactly five spaces. Thus, the format string for the labels (CITY_FMT_STR) should be "%5s", and the format string for the integer entries (TIME_FMT_STR) should be "%5d". Using named constants for format strings allows each format string to be used in many places, and it makes it easy and safe to alter them at any later time—just change the values assigned to the named constants at the beginning of the method.

In the displayFlightTimesTable method, note the three for loop headers. They all use the length property for their termination condition. Since length holds 5, the program would run correctly if you replaced the length termination conditions with hardcoded 5's. But don't do it. Using the length property makes the implementation more scalable. *Scalable* means that it's easy to change the amount of data that the program uses. For example, in the FlightTimes program, using a cities.length loop termination condition means that if you change the number of cities in the program, the program will still work properly.

Multidimensional Arrays

Arrays may have more than two dimensions. Arrays with three or more dimensions use the same basic syntax except they have additional [ ]'s. The first pair of brackets corresponds to the largest scale, and each subsequent pair of brackets nests within the previous pair, at progressively smaller levels of scale. For example, suppose the Missouri-Kansas airline company decides to go “green” and expands its fleet with new solar-powered airplanes and airplanes that burn hydrogen generated by wind turbines. The new airplanes have different flight times than the original jet-fuel airplanes. Thus, they need their own flight-times tables. The solution is to create a three-dimensional array where the first dimension specifies the airplane type—0 for the jet-fuel airplanes, 1 for the solar-powered airplanes, and 2 for the hydrogen-powered airplanes. Here’s how to declare the new three-dimensional flightTimes array instance variable:

```
private int[ ][ ][ ] flightTimes;
```

9.10 Arrays of Objects

You learned in the previous section

that a two-dimensional array is actually an array of references where each reference points to an array object. Now let's look at a related scenario. Let's look at an array of references where each reference points to a programmer-defined object. For example, suppose you'd like to store total sales for each sales clerk in a department store. If sales clerk Amanda sells two items for $55.45 and $22.01, then you'd like to store 77.46 as her total-sales value. You can store the sales clerk data in an array, clerks, where each element holds a reference to a SalesClerk object. Each SalesClerk object holds a sales clerk's name and the total sales for that sales clerk. See Figure 9.20 for an illustration of what we're talking about.

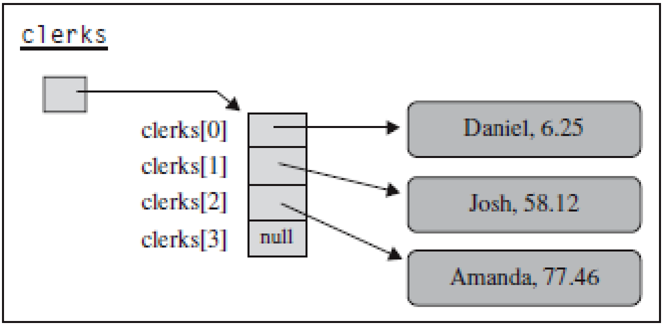

clerks
clerks[0]
clerks[1]
clerks[2]
clerks[3]
null
Daniel, 6.25
Josh, 58.12
Amanda, 77.46

Figure 9.20 Array of objects that stores sales-clerk sales data

The clerks array is an array of references. But most folks in industry would refer to it as an array of objects, and that's what we'll do too. An array of objects isn't that much different from an array of primitives. In both cases, you access each array element with square brackets (e.g., clerks[0], clerks[1]). But there are some differences you should be aware of, and those differences are the main focus of this section.

Need to Instantiate an Array of Objects and the Objects in That Array

With an array of primitives, you perform one instantiation—you instantiate the array object and that’s it. But with an array of objects, you have to instantiate the array object, and you must also instantiate each element object that’s stored in the array. It’s easy to forget the second step, the instantiation of individual element objects. If you do forget, then the elements contain default values of null, as illustrated by ⚠ clerks[3] in Figure 9.20. For the empty part of a partially filled array, null is fine, but for the part of an array that’s supposed to be filled, you need to overlay null with a reference to an object. The following is an example of how to create an array of objects—more specifically, how to create the clerks array of objects shown in Figure 9.20. Note the separate instantiations, with the new operator, for the clerks array and for each SalesClerk object.

```
SalesClerk[] clerks = new SalesClerk[4];
clerks[0] = new SalesClerk(”Daniel”, 6.25);
clerks[1] = new SalesClerk(”Josh”, 58.12);
clerks[2] = new SalesClerk(”Amanda”, 77.46);
```

Can’t Access Array Data Directly

With an array of primitives, you can access the array’s data, the primitives, directly. For example, the following code fragment shows how you can assign and print the first rainfall value in a rainfall array. Note how the value is directly accessed with rainfall[0]:

```
double[] rainfall = new double[365];
rainfall[0] = .8;
System.out.println(rainfall[0]);
```

On the contrary, with an array of objects, you normally cannot access the array’s data directly. The data is stored in variables inside the objects. Because the variables are normally private, you normally have to call a constructor or method to access them. For example, the following code fragment shows how you can use a constructor to assign Daniel and 6.25 to the first object in the clerks array. It also shows how you can use accessor methods to print the first object's name and sales data:

```
SalesClerk[] clerks = new SalesClerk[4];
clerks[0] = new SalesClerk("Daniel", 6.25);
System.out.println(
  clerks[0].getName() + ", " + clerks[0].getSales());
```

SalesClerks Program

Start with a UML class diagram to get a big-picture understanding.

Let's now implement a complete program that adds sales and prints sales for a group of sales clerks in a department store. As described in Section 8.8, we'll first get a big-picture view of things by presenting a UML class diagram. Figure 9.21's class diagram shows two classes. The SalesClerks class represents sales data for the entire department store, and the SalesClerk class represents total sales for one particular sales clerk.

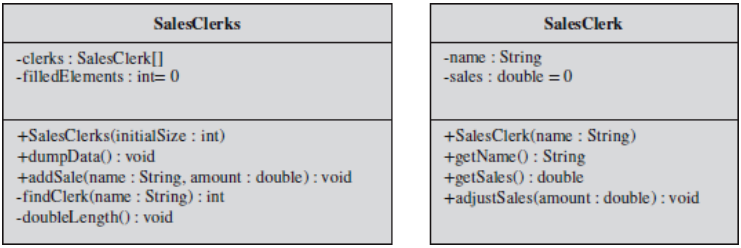

SalesClerks
-clerks : SalesClerk[]
-filledElements : int= 0
+SalesClerks(initialSize : int)
+dumpData() : void
+addSale(name : String, amount : double) : void
-findClerk(name : String) : int
-doubleLength() : void
SalesClerk
-name : String
-sales : double = 0
+SalesClerk(name : String)
+getName() : String
+getSales() : double
+adjustSales(amount : double) : void

Figure 9.21 UML class diagram for the SalesClerks program

The SalesClerks class contains two instance variables—clerks and filledElements. clerks is an array of SalesClerk objects. filledElements stores the number of elements that have been filled so far in the clerks array. For a filledElements example, see Figure 9.20, where filledElements would be 3. The SalesClerks's constructor instantiates the clerks array, using the constructor's initialSize parameter for the array's size.

The SalesClerks class in Figures 9.22a and 9.22b contains four methods—dumpData, addSale, findClerk, and doubleLength. The dumpData method is the most straightforward of the four. It prints all the data in the clerks array. The term *dump* is a computer term that refers to a simple (unformatted) display of a program's data. See the dumpData method in Figure 9.22b and verify that it prints the data in the clerks array.

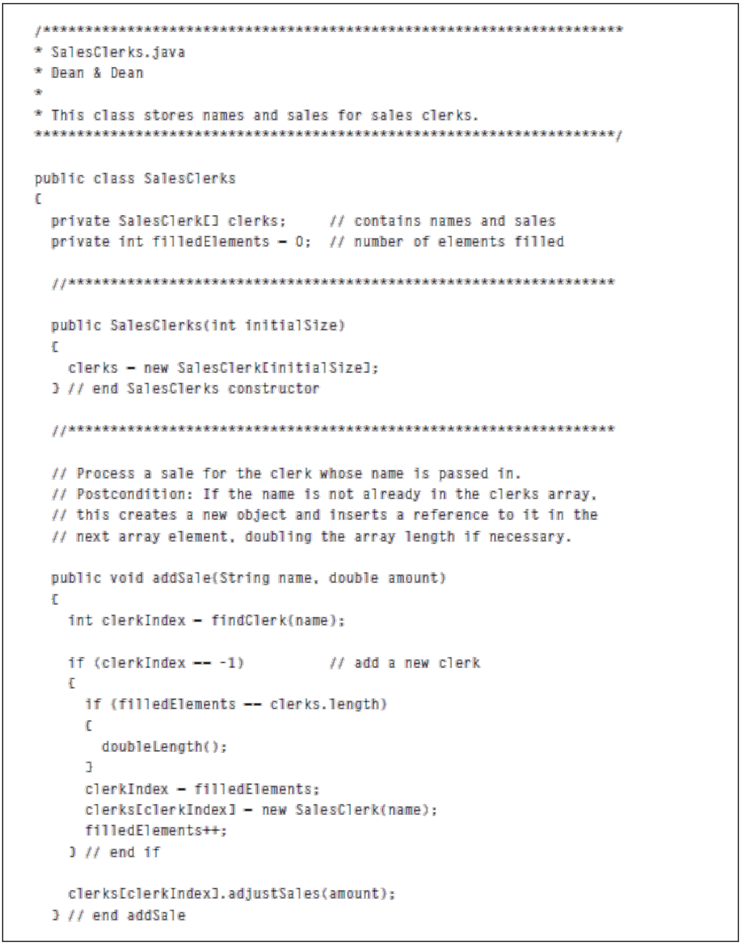
The addSale method processes a sale for a particular sales clerk. More specifically, the addSale method finds the sales clerk specified by its name parameter and updates that sales clerk's total sales with the value specified by its amount parameter. To find the sales clerk, the addSale method calls the findClerk helper method. The findClerk method performs a sequential search through the clerks array, and returns the index of the found sales clerk or −1 if the sales clerk is not found. If the sales clerk is not found, addSale adds a new

SalesClerk object to the clerks array in order to store the new sale transaction in it. In adding a new SalesClerk object to the clerks array, addSale checks to make sure that there is available space in the clerks array for the new SalesClerk object. If the clerks array is full (that is, filledElements equals clerks.length), then addSale must do something to provide for more elements. That's where the doubleLength helper method comes to the rescue.

The doubleLength method, as its name suggests, doubles the size of the clerks array. To do that, it instantiates a new array, clerks2, whose length is twice the length of the original clerks arrays. Then it copies all the data from the clerks array into the lowest-numbered elements in the clerks2 array. Finally, it assigns the clerks2 array to the clerks array so the clerks array points to the new longer array.

See the addSale, findClerk, and doubleLength methods in Figures 9.22a and 9.22b and verify that they do what they're supposed to do.

The SalesClerk class, shown on the right side of Figure 9.21, is fairly straightforward. It contains two instance variables, name and sales, for the sales clerk's name and the sales clerk's total sales. It contains two accessor methods, getName and getSales. It contains an adjustSales method that updates the sales clerk's total sales value by adding the passed-in amount to the sales instance variable. See the SalesClerk class in Figure 9.23 and verify that it does what it's supposed to do.

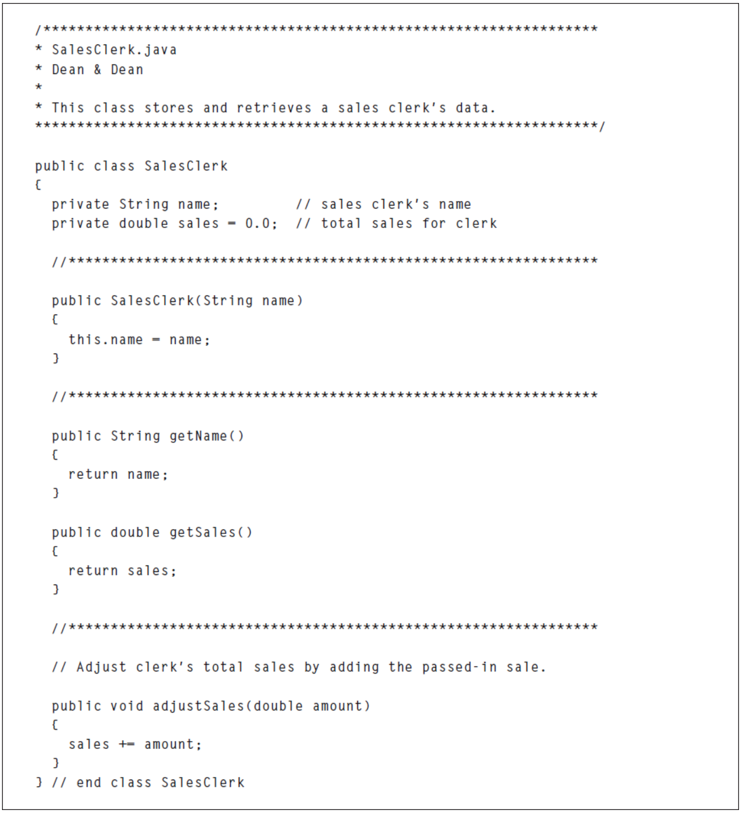


```
/*********************************************************************
* SalesClerks.java
* Dean & Dean
*
* This class stores names and sales for sales clerks.
*********************************************************************/

public class SalesClerks
{
  private SalesClerk[] clerks;     // contains names and sales
  private int filledElements = 0;  // number of elements filled

  //******************************************************************

  public SalesClerks(int initialSize)
  {
    clerks = new SalesClerk[initialSize];
  } // end SalesClerks constructor

  //******************************************************************

  // Process a sale for the clerk whose name is passed in.
  // Postcondition: If the name is not already in the clerks array,
  // this creates a new object and inserts a reference to it in the
  // next array element, doubling the array length if necessary.

  public void addSale(String name, double amount)
  {
    int clerkIndex = findClerk(name);

    if (clerkIndex == -1)          // add a new clerk
    {
      if (filledElements == clerks.length)
      {
        doubleLength();
      }
      clerkIndex = filledElements;
      clerks[clerkIndex] = new SalesClerk(name);
      filledElements++;
    } // end if

    clerks[clerkIndex].adjustSales(amount);
  } // end addSale
```

Figure 9.22a SalesClerks class—part A

```
  //***************************************************************

  // Print all the data - sales clerk names and sales.

  public void dumpData()
  {
    for (int i=0; i<filledElements; i++)
    {
      System.out.printf("%s: %6.2f\n",
        clerks[i].getName(), clerks[i].getSales());
    }
  } // end dumpData

  //***************************************************************

  // Search for the given name. If found, return the index.
  // Otherwise, return -1.

  private int findClerk(String name)
  {
    for (int i=0; i<filledElements; i++)
    {
      if (clerks[i].getName().equals(name))
      {
        return i;
      }
    } // end for
    return -1;
  } // end findClerk

  //***************************************************************

  // Double the length of the array.

  private void doubleLength()
  {
    SalesClerk[] clerks2 = new SalesClerk[2 * clerks.length];
    System.arraycopy(clerks, 0, clerks2, 0, clerks.length);
    clerks = clerks2;
  } // end doubleLength
} // end class SalesClerks
```

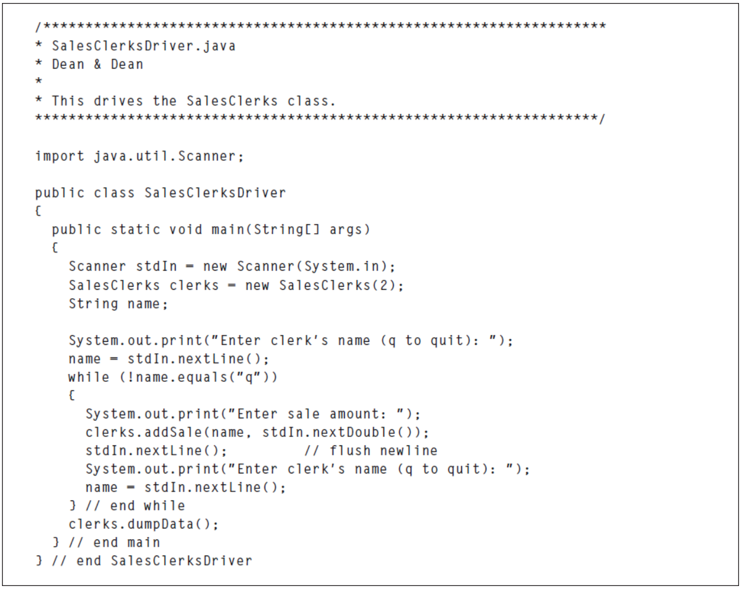
Figure 9.22b SalesClerks class—part B



```
/*******************************************************************
* SalesClerk.java
* Dean & Dean
*
* This class stores and retrieves a sales clerk's data.
*******************************************************************/

public class SalesClerk
{
  private String name;         // sales clerk's name
  private double sales = 0.0;  // total sales for clerk

  //****************************************************************

  public SalesClerk(String name)
  {
    this.name = name;
  }

  //****************************************************************

  public String getName()
  {
    return name;
  }

  public double getSales()
  {
    return sales;
  }

  //****************************************************************

  // Adjust clerk's total sales by adding the passed-in sale.

  public void adjustSales(double amount)
  {
    sales += amount;
  }
} // end class SalesClerk
```

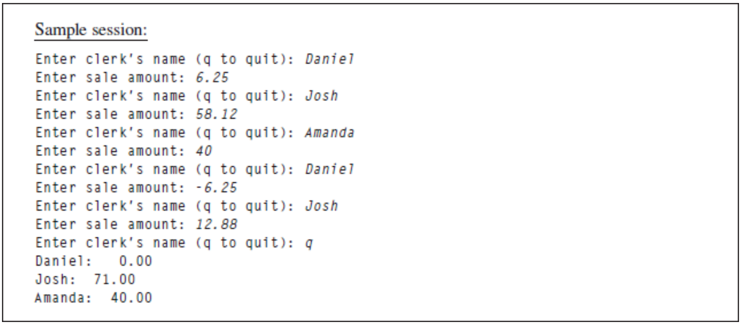
Figure 9.23 SalesClerk class

Now look at the main method in the Figure 9.24a's SalesClerksDriver class. A declaration instantiates a SalesClerks object, passing an initial array-length value of 2 to the SalesClerks constructor. Then it repeatedly prompts the user for a sales-clerk name and sales value and calls addSale to insert the input data into the SalesClerks object. The looping stops when the user enters a q for the next name. Then main calls dumpData to display the accumulated sales data. Figure 9.24b shows the output.



```
/*******************************************************************
* SalesClerksDriver.java
* Dean & Dean
*
* This drives the SalesClerks class.
*******************************************************************/

import java.util.Scanner;

public class SalesClerksDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    SalesClerks clerks = new SalesClerks(2);
    String name;

    System.out.print("Enter clerk's name (q to quit): ");
    name = stdIn.nextLine();
    while (!name.equals("q"))
    {
      System.out.print("Enter sale amount: ");
      clerks.addSale(name, stdIn.nextDouble());
      stdIn.nextLine();         // flush newline
      System.out.print("Enter clerk's name (q to quit): ");
      name = stdIn.nextLine();
    } // end while
    clerks.dumpData();
  } // end main
} // end SalesClerksDriver
```

Figure 9.24a Driver for the SalesClerks program in Figures 9.22a, 9.22b, and 9.23



Sample session:

```
Enter clerk's name (q to quit): Daniel
Enter sale amount: 6.25
Enter clerk's name (q to quit): Josh
Enter sale amount: 58.12
Enter clerk's name (q to quit): Amanda
Enter sale amount: 40
Enter clerk's name (q to quit): Daniel
Enter sale amount: -6.25
Enter clerk's name (q to quit): Josh
Enter sale amount: 12.88
Enter clerk's name (q to quit): q
Daniel:    0.00
Josh:  71.00
Amanda:  40.00
```

Figure 9.24b SalesClerks program output

9.11 For-Each Loops

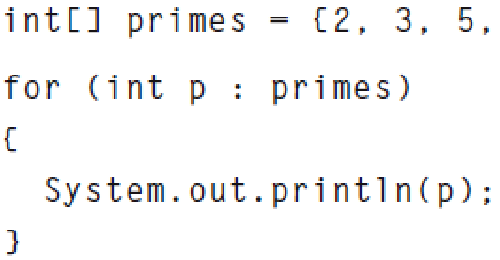
A *for-each loop* (sometimes called an *enhanced for loop*) is a modified version of the traditional for loop. It provides a handy way to iterate through all the elements in an array when you don't know or don't care exactly where particular elements are located. Here is the for-each loop syntax:

```
for (element-type element-name : array-reference-variable)
{
  . . .
}
```

Are you wondering why the for-each

loop is called a for-each loop, even though there's no "each" in the syntax? It's because most people say "for each" to themselves when reading a for-each loop's header.

Consider, for example, the following code fragment, which prints all the numbers in a given array of prime numbers:

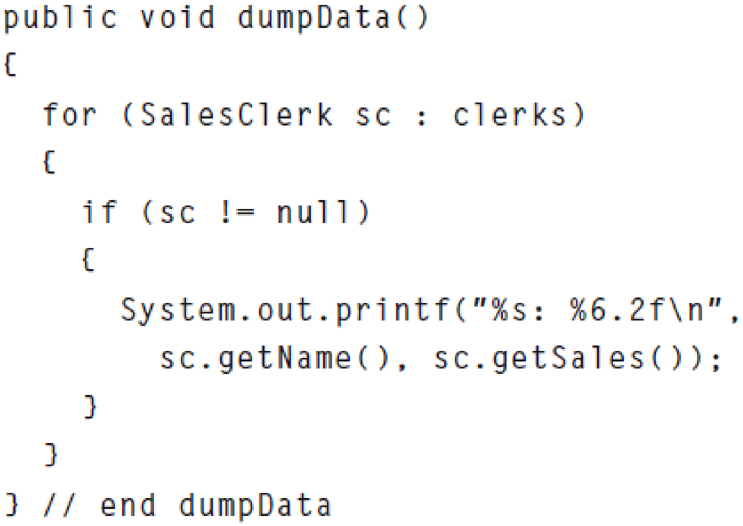


```
int[] primes = {2, 3, 5, 7, 11, 13};

for (int p : primes)
{
  System.out.println(p);
}
```

When we look at this for-each loop, we say to ourselves, "For each p in primes, print p." That verbal expression explains the "for-each" terminology.

Note how the for-each loop header conforms to the previous syntax. The primes reference variable refers to the whole array, and the p variable holds a typical element in that array. It's legal to choose any name for the element. Normally, good Java style dictates use of longer descriptive names for variables, and that rule says we should have used a descriptive word like prime for the variable that refers to a typical element in this example. But because the for-each loop header is so self-explanatory, it’s accepted practice to abbreviate the for-each element name in a for-each loop, just as it’s accepted practice to abbreviate the index in an ordinary for loop. Because it’s accepted, common practice, we show it here. But go ahead and use a longer word whenever you feel the urge. Descriptive words do improve understanding, and when for-each loops are large, we prefer longer words instead of abbreviations.

As another example, consider this alternative implementation of the dumpData method in the SalesClerks class in Figure 9.22b. It uses a for-each loop to step through the elements in the clerks array:

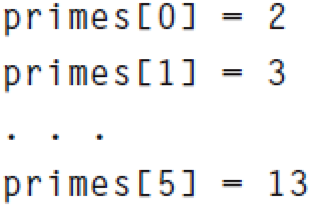


```
public void dumpData()
{
  for (SalesClerk sc : clerks)
  {
    if (sc != null)
    {
      System.out.printf("%s: %6.2f\n",
        sc.getName(), sc.getSales());
    }
  }
} // end dumpData
```

When you look at the header in this particular for-each loop, say to yourself, “for each SalesClerk in clerks, . . .”.

A for-each loop avoids index initialization, testing, and incrementing. It also simplifies element method calls. In this example, we had to put the printf statement inside an if statement to avoid an error when the iteration got beyond the filledElements limit. At first, it looks like this added complexity offsets the simplifications. But what if one of the existing clerks quits, and we remove that clerk’s name from the array? An ordinary for loop would generate an error when the iteration hit that null cell in the array. So, actually, the null test we added to make dumpData work with a for-each loop should also be present in an ordinary for loop. The next chapter includes an array alternative (an ArrayList) that automatically expands and shrinks with element additions and deletions. This makes it possible to use a for-each loop without an explicit -null-element test.

The for-each loop is great, but you should be aware of several issues when using it: (1) It was introduced in Java 5.0, so it won’t work with older compilers. (2) The for-each loop doesn’t use an index variable to loop through its elements. That can be a benefit in that it leads to less cluttered code. But it’s a drawback if there’s a need for an index within the loop. For example, suppose you have an initialized primes array, and you want to display it like this:



```
primes[0] = 2

primes[1] = 3

. . .

primes[5] = 13
```

The numbers inside the square brackets are index values. So if you implemented a solution with a for-each loop, you'd have to add an index variable to your code and increment it each time through the loop. On the other hand, if you implemented a solution with a traditional for loop, you'd already have an incrementing index variable built in.

Summary

- Arrays facilitate the representation and manipulation of collections of similar data. You access array elements with *array-*

name[index], where index is a nonnegative integer, starting at zero.

- You can create and completely initialize an array in one statement, like this:

  element-type[] *array-name* = {*element0*, *element1*, ...};

- Usually, however, it's more useful to defer element initialization and use new to create an array of uninitialized elements, like this:

  element-type[] *array-name* = new *element-type*[*array-size*];

- You can read or write directly to an array element by inserting an appropriate index value in square brackets after the array name at any time after the array has been created.

- Every array automatically

includes a public property called length, which you can access directly with the array name. The highest index value is a*rray-name*.length - 1.

- To copy an array, copy each of its elements individually, or use the System.arraycopy method to copy any subset of elements in one array to any location in another array.
- A histogram is an array of elements in which each element's value is the number of occurrences of some event.
- A sequential search is a good way to search for a match in an array whose length is less than about 20, but for long arrays, you should first sort the array with the Arrays.sort method and then use a binary search.
- A two-dimensional array is an

array of arrays, declared with two sets of square brackets after the -element-type identification. You can instantiate it with an initializer or with new followed by element type and two array-size specifications in square brackets.

- In creating an array of objects, multiple instantiations are required. After instantiating the array, you also need to instantiate the individual element objects within the array.
- Use a for-each loop when you don’t know or don’t care exactly where the element is.

Review Questions

§9.2 Array Basics

1. It’s legal to store ints and also

doubles in a single standard array. (T / F)

2. Given an array that's named myArray, you access the first element in the array using myArray[0]. (T / F)

§9.3 Array Declaration and Creation

3. Provide a declaration for an array of strings called names.

4. Consider the heading for any main method:

> public static void main(String[] args)

What kind of a thing is args?

5. Suppose you create an array with the following statement:

> int[] choices = new int[4];

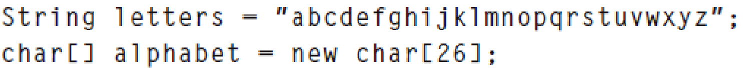
What is the default value in a typical element of this array? Is it garbage or something in particular?

§9.4 Array length Property and Partially Filled Arrays

6. The value of an array's length equals the value of the array's largest acceptable index. (T / F)

§9.5 Copying an Array

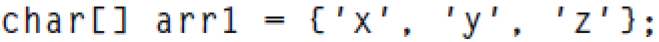
7. Given



```
String letters = "abcdefghijklmnopqrstuvwxyz";
char[] alphabet = new char[26];
```

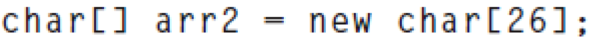
Write a for loop that initializes alphabet with the characters in letters.

8. Write a single statement that copies all the elements in



```
char[] arr1 = {'x', 'y', 'z'};
```

to the last three elements of



```
char[] arr2 = new char[26];
```

§9.6 Problem Solving with Array Case Studies

9. In Figure 9.7's MovingAverage

program, suppose you want to shift in the other direction. How would you write the inner for loop header, and how would you write the array assignment statement in the inner for loop?

10. What kind of value does a typical histogram “bin” contain?

§9.7 Searching an Array

Page 428

11. It’s possible to search array ids for an element equal to id with nothing more than this:

```
int i;
for (i=0; i<ids.length && id != ids[i]; i++)
{ }
if (<boolean expression>)
{
  return i;
}
```

What is the <*boolean expression*> that indicates that i has been found?

§9.8 Sorting an Array

12. We elected to use static methods to implement our sort algorithm. What is an advantage of that?

13. Java's API sort method is in what class?

§9.9 Two-Dimensional Arrays

14. We have said that a two-dimensional array is an array of arrays. Consider the following declaration:

> double[][] myArray = new double[5][8];

In the context of the expression, array of arrays, what does myArray[3] mean?

§9.10 Arrays of Objects

15. In creating an array of objects, you have to instantiate the array object, and you must also instantiate each element object that's stored in the array. (T / F)

§9.11 For-Each Loop

16. You must use a for-each loop, and not a traditional for loop, whenever you need to iterate through a collection of elements. (T / F)

Exercises

---

1. [after §9.2] The index number of the last element in an array of length 60 is ______________.

2. [after §9.3] Declare an array named zipCodes that holds int values.

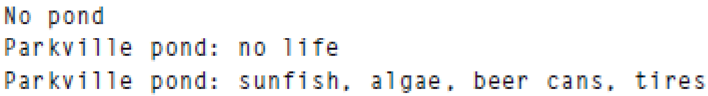
3. [after §9.3] Provide a single initialization statement that initializes all the elements in a 4-element array called ids to −1, where ids is an array of long elements.

4. [after §9.4] Pond Program:

As part of your internship at Parkville’s Parks and Recreation Department, you’ve been asked to write a program that keeps track of pond life. Your program doesn’t have to do much—it simply prints pond life forms. For this exercise, provide a Pond class that uses the following two instance variables and supports the subsequent main method:

```
private String city;
private String[] life = new String[] {"no life"};

public static void main(String[] args)
{
  Pond pond1 = new Pond();
  Pond pond2 = new Pond("Parkville", null);
  Pond pond3 = new Pond("Parkville",
    new String[] {"sunfish", "algae", "beer cans", "tires"});

  pond1.display();
  pond2.display();
  pond3.display();
} // end main
```

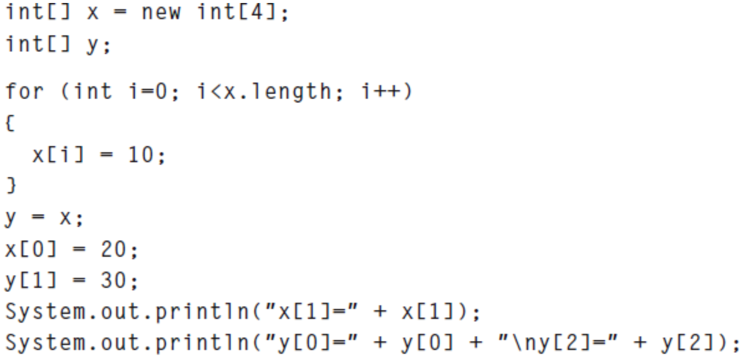
When run, the main method should print this:



```
No pond
Parkville pond: no life
Parkville pond: sunfish, algae, beer cans, tires
```

Although it’s not required, you’re encouraged to write a complete program in order to test your Pond class.

5. [after §9.5] Assume that this code fragment compiles and runs. What is its output? Be precise when showing your output.



```
int[] x = new int[4];
int[] y;

for (int i=0; i<x.length; i++)
{
  x[i] = 10;
}
y = x;
x[0] = 20;
y[1] = 30;
System.out.println(”x[1]=” + x[1]);
System.out.println(”y[0]=” + y[0] + ”\ny[2]=” + y[2]);
```

6. [after §9.5] Given the following allPrices initializer, provide a declaration statement for an suvPrices array such that the new array is half the size of the allPrices array. You can assume that there are an even number of elements in the allPrices array. Provide an arraycopy method call that copies half of the values—the smaller-index values—from allPrices to suvPrices.

```
double[] allPrices = {10000, 11000, 25000, 18000,
  30000, 9000, 12000, 21000};
```

7. [after §9.5] The following program is supposed to reverse the elements in the scientists array. It compiles and runs, but it doesn't work properly.

```
import java.util.Arrays;

public class Reverse
{
  public static void main(String[] args)
  {
    String[] scientists = {"Sheldon", "Amy", "Raj"};
    reverse(scientists);
    System.out.println(
      scientists[0] + " " + scientists[1] + " " + scientists[2]);
  } // end main
  public static void reverse(String[] list)
  {
    String[] temp = Arrays.copyOf(list, list.length);
    for (int i=0; i<list.length; i++)
    {
      temp[i] = list[list.length-i-1];
    }
    list = temp;
  } // end reverse
} // end class Reverse
```

Sample session:

```
Sheldon Amy Raj
```

The reversal does not occur. Fix

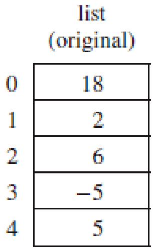
the problem by rewriting the reverse method. Hint: You’ll need to modify one line of code. Although not required, to reduce clutter, you should remove another line of code.

8. [after §9.6] Write a program that performs a one-position *left rotation* of the elements in an array. Demonstrate the effect by having the program rotate the elements in this particular array:

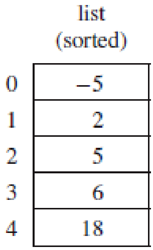
```
String[] scientists = {"Sheldon", "Amy", "Raj"};
```

The rotation should change the sequence of the array’s elements to: “Amy,” “Raj,” and “Sheldon.” Then it should print the new sequence. Your solution should perform the rotation without creating another array, although you will need a variable to temporarily hold an element’s value.

9. [after §9.8] Given the following list array, use the selection sort algorithm to sort the array. Show each step of the selection sort process. Do not provide code; just show pictures of the list array after each element swap.



	list (original)
0	18
1	2
2	6
3	–5
4	5



	list (sorted)
0	–5
1	2
2	5
3	6
4	18

10. [after §9.8] The insertion-sort algorithm provides an alternative to the selection-sort algorithm for sorting something like a hand of cards, where there is a small number of items (about 20 or less). It’s more efficient than selection sort if the array is only slightly out of order, but it’s relatively inefficient for large numbers of items. The following code implements a descending insertion-sort algorithm:

```
1   public static void insertionSort(int[] cards)
2   {
3     int pick;
4     int j;
5
6     for (int i=1; i<cards.length; i++)
7     {
8       pick = cards[i];       // pick each successive element
9       // shift all previously sorted smaller elements
10        for (j=i; j>0 && pick>cards[j-1]; j--)
11        {
12          cards[j] = cards[j-1];
13        }
14        cards[j] = pick;       // insert as next highest
15      }
16    } // end insertionSort
```

Note that the scope of the j count variable extends beyond the scope of the for loop in which it's used. Assume that an array of int has been instantiated and the insertionSort method has been called with a reference to this array passed in as a parameter. Trace the execution of this method, using the following header and initial entries:

Sort					*<arrays>*					
	insertionSort				arr1					
line#	(cards)	i	j	pick	length	0	1	2	3	4
					5	3	2	6	9	5
1	arr1									

11. [after §9.8] Write a program

that uses the sort method in the Java Arrays class to sort the elements in an array of String elements in alphabetical order. Use this array:

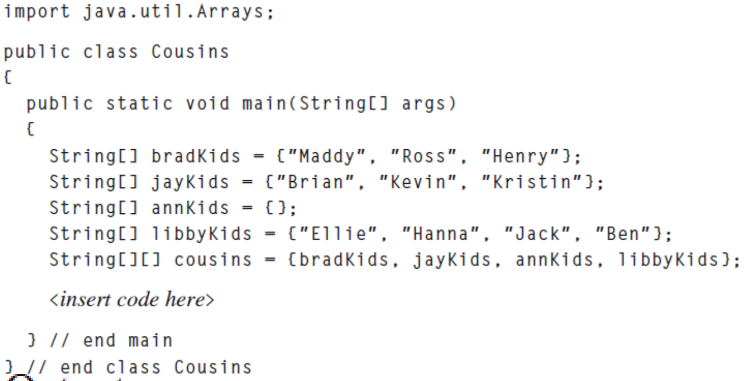
```
String[] friends =
  {"Sheldon", "Amy", "Leonard", "Bernadette", "Raj"};
```

Sample session:

```
Amy Bernadette Leonard Raj Sheldon
```

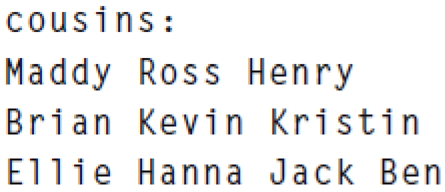
12. [after §9.9] Write a single statement that initializes a two-dimensional array named rectangles with double elements. Provide one row for each of three rectangles and give each row two columns, one for width and the other for height. Initialize all width and height values to 1.0.

13. [after §9.9] As you learned earlier, a two-dimensional array is actually a one-dimensional array of references to other (subordinate) one-dimensional arrays, which may be of different lengths. The following program is supposed to print the contents of the cousins array. Replace the <*insert code here*> line with appropriate code.



```
import java.util.Arrays;

public class Cousins
{
  public static void main(String[] args)
  {
    String[] bradKids = {"Maddy", "Ross", "Henry"};
    String[] jayKids = {"Brian", "Kevin", "Kristin"};
    String[] annKids = {};
    String[] libbyKids = {"Ellie", "Hanna", "Jack", "Ben"};
    String[][] cousins = {bradKids, jayKids, annKids, libbyKids};

    <insert code here>

  } // end main
} // end class Cousins
```

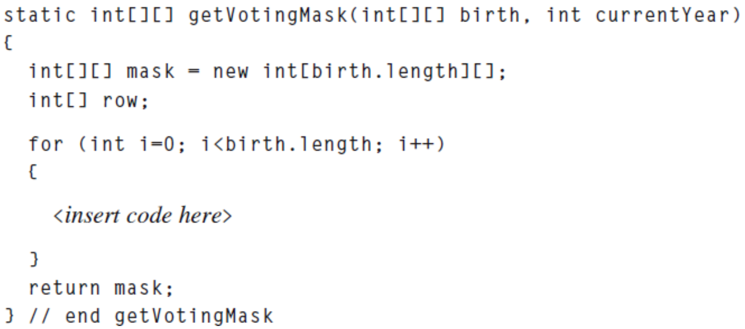
Output:



```
cousins:
Maddy Ross Henry
Brian Kevin Kristin
Ellie Hanna Jack Ben
```

Page 432

14. [after §9.9] The programming term *mask* refers to an array or 0's and 1's that is built from another array, such that if any element in the other array satisfies some criterion, the corresponding mask element is 1. Otherwise that corresponding mask element is 0. Suppose each row in a two-dimensional array named births contains the birth years of children in one family. And suppose we want a method named getVotingMask to return a mask whose 1 or 0 elements indicate whether a child is at or above age 18, the legal age to vote in America. The getVotingMask method's first parameter is a reference to the births array and its second parameter is the current year. We want the method to return a mask whose elements are 0 if the corresponding child is under 18 or a 1 if the corresponding child is 18 or over. Supply the missing code in the following getVotingMask method:



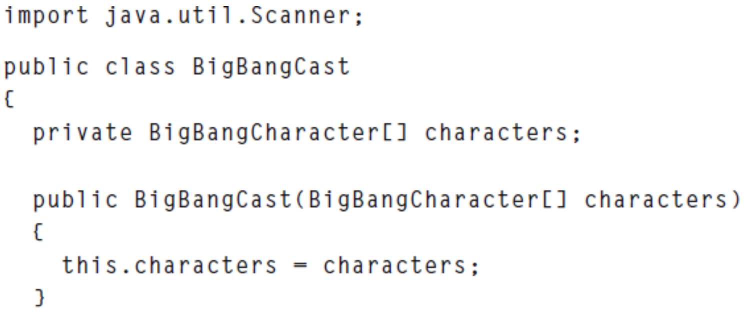
```
static int[][] getVotingMask(int[][] birth, int currentYear)
{
  int[][] mask = new int[birth.length][];
  int[] row;

  for (int i=0; i<birth.length; i++)
  {

    <insert code here>

  }
  return mask;
} // end getVotingMask
```

15. [after §9.10] Modify the SalesClerks program so the SalesClerk array length always equals the current number of sales clerks. That means the initial length of the SalesClerks array should be zero. Because the current number of sales clerks always equals clerks.length, dispense with the filledElements variable. Also dispense with the doubleLength method, and do the job of incrementing the length of the clerks array in the addSale method, mostly within the block under:

if (clerkIndex = = -1}.

16. [after §9.10] Given the

following program skeleton that stores information about the characters in the *Big Bang Theory* television show. Replace the <*insert displayActor code here*> line with code that implements a displayActor method. The method receives a character's name and searches for the name in the array of BigBangCharacter objects. If the character is found, display the actor who plays the character. Otherwise, print that the character wasn't found. See the sample sessions for output format details.



```
import java.util.Scanner;

public class BigBangCast
{
  private BigBangCharacter[] characters;

  public BigBangCast(BigBangCharacter[] characters)
  {
    this.characters = characters;
  }
  <insert displayActor code here>

  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String character;
    BigBangCharacter[] characters =
      {new BigBangCharacter("Bernadette", "Melissa Rauch"),
       new BigBangCharacter("Raj", "Kunal Nayyar"),
       new BigBangCharacter("Amy", "Mayim Bialik"),
       new BigBangCharacter("Sheldon", "Jim Parsons"),
       new BigBangCharacter("Howard", "Simon Helberg"),
       new BigBangCharacter("Leonard", "Johnny Galecki")};
    BigBangCast cast = new BigBangCast(characters);
    System.out.print("Enter a Big Bang Theory character: ");
    character = stdIn.nextLine();
    cast.displayActor(character);
  } // end main
} // end class BigBangCast

public class BigBangCharacter
{
  private String name;
  private String actor;

  public BigBangCharacter(String name, String actor)
  {
    this.name = name;
    this.actor = actor;
  }

  public String getName()
  {
    return this.name;
  }

  public String getActor()
  {
    return this.actor;
  }
} // end class BigBangCharacter
```

Sample session:

```
Enter a Big Bang Theory character: Amy
Amy is played by Mayim Bialik.
```

Another sample session:

```
Enter a Big Bang Theory character: Trevor Noah
Sorry - couldn't find Trevor Noah in the list of characters.
```

17. [after §9.11] In this chapter's FlightTimes class, modify the displayFlightTimesTable method by converting the two for loops that iterate over the columns, so that they use for-each loops instead.

Note: There's no comparable benefit to converting the for loop that iterates over the rows. Why? Because if it's converted to a for-each loop, then there's no row index variable, and that variable is needed later on as an index into the cities array:

System.out.printf(CITY_FMT_STR, cities[row]);

Review Question Solutions

---

1. False. The types of the data elements in a particular array must be the same.

2. True.

3. Declaration for an array of strings called names:

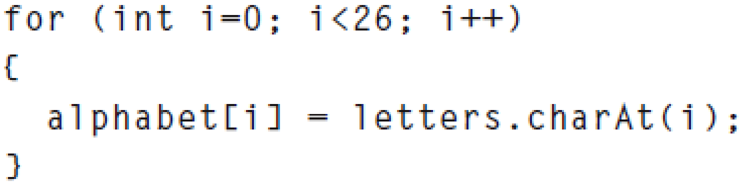
String[] names;

4. The args parameter in main is an array of strings.

5. The elements of an array are like the instance variables in an object. Array-element default values are not garbage. The default value of an int [] element is 0.

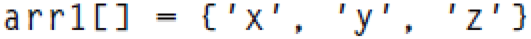
6. False. The largest acceptable index value is one less than the array's length.

7. This code fragment initializes the character array, alphabet:



```
for (int i=0; i<26; i++)
{
  alphabet[i] = letters.charAt(i);
}
```

8. You can copy:



```
arr1[] = {'x', 'y', 'z'}
```

to the end of:

```
arr2[] = new char[26]
```

with the following statement:

```
System.arraycopy(arr1, 0, arr2, 23, 3);
```

9. In the MovingAverage program, to shift in the other direction, the inner for loop header is:

```
for (int d=days.length-1; d>0; d--)
```

The array element assignment statement in this loop is:

```
days[d] = days[d-1];
```

10. A histogram “bin” contains the number of occurrences of an event.

11. The Boolean expression that indicates that i has been found is:

```
(ids.length != 0 && i != ids.length)
```

12. The advantage of using static methods is that the sort method can be used with any passed-in array, not just on a specific instance variable array.

13. Java's API sort method is in the Arrays class.

14. myArray[3] refers to the fourth row, which happens to be an array of eight double values.

15. True.

16. False. You can use a traditional for loop (or a for-each loop) to iterate through a collection of elements.

---

[1] DFLA = dumb four-letter acronym.

ArrayLists and an Introduction to the Java Collections Framework

Objectives

- See how the ArrayList class makes arrays more flexible.
- Understand autoboxing.
- Pass anonymous objects to and

from methods.

- Appreciate the ease of ArrayList element insertion and deletion.
- Understand the LinkedList structure.
- Compare the performance of alternative software implementations.
- Learn how the Java API groups method interfaces to organize behavioral descriptions.
- Use Java API implementations of lists, queues, stacks, sets, and maps.
- Model a network and simulate flow through that network.

Outline

10.1 Introduction 436

10.2 The ArrayList Class 437

10.3 Storing Primitives in an ArrayList 443

10.4 ArrayList Example Using Anonymous Objects and the For-Each Loop 446

10.5 ArrayLists Versus Standard Arrays 450

10.6 The LinkedList Class 451

10.7 The List Interface 452

10.8 Problem Solving: How to Compare Method Execution Times 453

10.9 Queues, Stacks, and the ArrayDeque Class 457

10.10 Overview of the Java Collections Framework 464

10.11 Collections Example—Information Flow in a Network of Friends 468

10.12 GUI Track: Second

Iteration of Problem Solving with CRC Cards (Optional) 476

10.1 Introduction

An object stores a collection of related data, where the data can be of different types. An array stores a collection of related data, where the data must be of the same type. In this chapter, you’ll learn about other ways to store collections of related data of the same type.

We start by describing the Java API ArrayList class. Like an array, an ArrayList stores an ordered list of related data of the same type. But unlike an array, an ArrayList grows and shrinks dynamically as you add and remove elements to and from it. That can be a tremendous benefit when you don’t know the number of elements ahead of time. ArrayLists are able to store objects (actually, references to objects), but they are not able to store primitive values (e.g., ints and doubles) directly. In this chapter, we discuss how to overcome this limitation by wrapping primitive values in their respective wrapper classes before adding them to an ArrayList.

Discussion of ArrayLists takes about a third of this chapter. Later chapters use ArrayLists extensively, so make sure that you have a good grasp of them before moving on. The ArrayList class is part of the Java collections framework, which is a Java API library of classes that store groups of related data. The remainder of this chapter introduces other classes that are in the Java collections framework, as well as additional related topics. Later chapters do not rely on that material, so you'll be OK if you decide to skip over it. But if you skip over it now, you should come back to it later when you have more time. It’s useful stuff!

After ArrayLists, the chapter introduces another data structure—a *linked list*. Like arrays and ArrayLists, linked lists store ordered lists of related data of the same type, but a linked list uses a chain of references to connect its elements. In some programming languages, the strategy of connecting a list’s elements with a chain of references can lead to more efficient insertions and deletions in the middle of the list. To see if that holds true with Java, this chapter compares the two Java API classes, LinkedList and ArrayList, in terms of how fast they perform various operations like accessing, inserting, and removing list elements. You’ll see that ArrayList is usually significantly faster than LinkedList.

Next, the chapter describes the List *interface*. We’ll have more to say about interfaces in Chapter 14, but for now, just think of an interface as a template for designing classes that share certain qualities. The ArrayList and LinkedList classes both implement lists, and, as such, they share certain qualities common to all lists. Thus, to help with consistency, they are built using the List interface as a template.

ArrayLists, linked lists, and the List interface are rather general-purpose in nature, while two other data structures, *queues* and *stacks*, have narrower focus. Like the other data structures mentioned so far, queues and stacks store ordered lists of related data of the same type. But unlike the other data structures, queues and stacks are restricted in terms of which elements are manipulated. With a queue, you add elements to the back end and you remove elements from the front end. With a stack, you add elements and remove elements only from one end, which is called the *top*. After describing queue and stack concepts, the chapter presents programs that implement queues and stacks using the Java API ArrayDeque class.

Next, the chapter describes quite a few (but far from all) of the classes and interfaces in the Java collections framework and uses a picture to illustrate how they are connected. The next-to-last section in this chapter presents a complete program that illustrates how to use some of the more important Java collection framework classes and interfaces. We hope you'll find this program's subject matter—networks—to be engaging, because networks are ubiquitous.

The final section illustrates how to use Java Collections Framework interfaces and classes (Map and LinkedHashMap and Set and LinkedHashSet) in a GUI context. It provides another example of using ChoiceDialog and TextInputDialog classes introduced in Section 3.25. It shows how a JavaFX GUI program retrieves command prompt arguments from the Application class's Parameters variable, and it illustrates use of the PrinterJob class.

10.2 The ArrayList Class

As you learned in the previous chapter, arrays allow you to work with an ordered list of related data of the same type. Arrays work great for many lists, but if you have a list where the number of elements is hard to predict, they don't work so well. If you don't know the number of elements, you have to either (1) start with an array size that's large enough to accommodate the possibility of a very large number of elements or (2) create a new larger array whenever the array becomes full and you need more room for more elements. The first solution is wasteful of computer memory as it requires allocating space for a large array where most of the elements are unused. The second solution is what we did in the doubleLength method in the SalesClerks class in Figure 9.22b. It works OK in terms of saving memory, but it requires the programmer to do extra work (writing the code that creates a larger array).

To help with lists where the number of elements is hard to predict, Java language designers came up with the ArrayList class. The ArrayList class is built using an array, but the array is hidden in the background, so you can't access it directly. With an array in the background, the ArrayList class is able to provide the basic functionality that comes with a standard array. With its methods, the ArrayList class is able to provide additional functionality that helps when you don't know the number of elements. In this section, we discuss how to create an ArrayList and how to use its methods.

How to Create an ArrayList

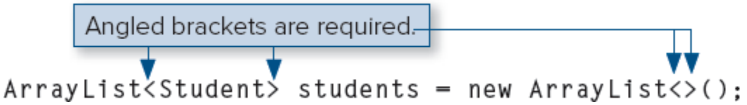
The ArrayList class is defined in the Java API's java.util package, so to use the class, you should provide an import statement, like this:

import java.util.ArrayList;

To initialize an ArrayList reference variable, use this syntax:

ArrayList<*element-type*> *reference-variable* = new ArrayList<>();

Note the angled brackets around *element-type*. The angled brackets are part of the required syntax. As indicated by the italics, *element-type* and *reference-variable* are descriptions. You should replace *element-type* with the type for the ArrayList's elements. You should replace *reference-variable* with an actual reference variable. For example, suppose you've defined a Student class, and you want an ArrayList of Student objects. Here's how to create such an ArrayList, named students:



Angled brackets are required.

```
ArrayList<Student> students = new ArrayList<>();
```

In the code above, the angled brackets on the right side form the *diamond operator*. The diamond operator is named as such because the two empty angled brackets look like a diamond turned on its side (< >). The diamond operator asks the compiler to perform *type inferencing*, which means the compiler determines the type of the instantiated ArrayList's elements by looking at prior code for clues. In the example above, the "clue" is found in the ArrayList declaration at the left, which shows that each element in the

ArrayList needs to be of type Student. You might be asking yourself, “How good is the diamond operator at finding and applying such clues?” The diamond operator’s abilities are actually fairly limited. The diamond operator works only for constructor calls, and the type inferencing must be obvious from the program’s context. For clarity, Oracle recommends that you use the diamond operator only if the constructor call is part of a variable declaration initialization statement.

The angled brackets for the diamond operator and for the declaration type are indicative of a *generic class*, or more generally a *generic type*. When using a generic class, you are required to specify one or more *type parameters*, where the type parameters serve as the types for variables declared within the class’s definition. In the case of an ArrayList, its type parameter specifies the type for the ArrayList's elements. Generic classes are implemented such that they can support various classes for their type parameters. In the case of an ArrayList, it supports using the wrapper classes for its type parameter.

Besides the angled brackets, there are two additional noteworthy items in the above example. First, there is no size specification. That's because ArrayList objects start out with no elements and they automatically expand to accommodate however many elements are added to them. Second, the element type, Student, is a class name. For ArrayLists, you must specify a class name, not a primitive type, for the element type. Specifying a class name means that ArrayLists can hold only references to objects. They cannot hold primitives, like int or double. That's technically true, but there's an easy way to mimic storing primitives in an ArrayList. We'll discuss how to do that in the next section.

As explained above, you can use the diamond operator when instantiating an ArrayList as part of an initialization. On the other hand, if you instantiate a new ArrayList later and assign it into a previously declared ArrayList reference variable, you should instantiate your ArrayList with angled brackets around the ArrayList's element type, like this:

```
ArrayList<String> students;
. . .
students = new ArrayList<String>();
```

In the assignment above, if you use a diamond operator (i.e., use <> instead of <String>), the compiler might generate an error (due to type inferencing failing to work).

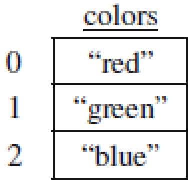
Adding Elements to an ArrayList

To convert an instantiated empty

ArrayList into something useful, you need to add elements to it. To add an element to the end of an ArrayList, use this syntax:

ArrayList-reference-variable.add (*item*);

The *item* that's added must be the same type as the element type specified in the ArrayList's declaration. Perhaps the simplest type of element object is a string, so let's start with an ArrayList of strings. Suppose you want to write a code fragment that creates this ArrayList object:



Try writing the code on your own before proceeding. When you're done, compare your answer to this:

```
import java.util.ArrayList;
. . .
ArrayList<String> colors = new ArrayList<>();
colors.add("red");
colors.add("green");
colors.add("blue");
```

The order in which you add elements determines the elements' positions. Because we added "red" first, it's at index position 0. Because we added "green" next, it's at index position 1. Likewise, "blue" is at index position 2.

API Headings

In describing the ArrayList class, we'll use *API headings* to present the ArrayList class's methods. As you may recall from Chapter 5, API stands for application programming interface, and API headings are the source code headings for the methods and constructors in Java's library of prebuilt Java classes. The API headings tell you how to use the methods and constructors by showing you their parameters and return types. For example, here's the API heading for the Math class's pow method:

```
public static double pow(double num, double power)
```

The above line tells you everything you need to know to use the pow method. To call the pow method, pass in two double arguments: one argument for the base and one argument for the power. The static modifier tells you to preface the call with the class name and then a dot. The double return value tells you to embed the method call in a place that can use a double value. Here's an example that calculates the volume of a sphere:

```
double volume = (4.0 / 3) * Math.PI * Math.pow(radius, 3)
```

How to Access Elements in an

ArrayList

With standard arrays, you use square brackets to read and update an element. But with an ArrayList you don't use square brackets. Instead, you use a get method to read an element's value and a set method to update an element's value.

Here's the API heading for the ArrayList's get method:

```
public E get(int index)
```

The index parameter specifies the position of the desired element within the ArrayList calling object. For example, the following method call retrieves the second element in a colors ArrayList:

```
colors.get(1);
```

If the index parameter refers to a nonexistent element, then a runtime error occurs. For example, if colors contains three elements, then this generates a runtime error:

```
colors.get(3);
```

In the get method's API heading, note the E return type:

```
public E get(int index)
```

The E stands for "element." It represents the data type of the ArrayList's elements, whatever that data type happens to be. So if an ArrayList is declared to have String elements, then the get method returns a String value, and if an ArrayList is declared to have Student elements, then the get method returns a Student value. The E in the get method's heading is a generic name for an element type. Using a generic name for a type is an important concept that will come up again with other methods.

Using a generic name for a type is possible with ArrayLists because the

ArrayList class is defined to be a *generic class*, by using <E> in its class heading:

public class ArrayList<E>

You don't need to understand generic class details in order to use ArrayLists, but if you want such details, visit https://docs.oracle.com/javase/tutorial/java/generics/.

How to Update an ArrayList Element

Now for the get method's partner, the set method. The set method allows you to assign a value to an ArrayList element. Here is the API heading for ArrayList's set method:

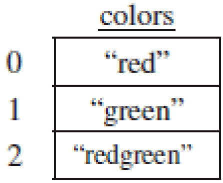
public E set(int index, E elem)

In the set method's API heading, the index parameter specifies the position of the element you're interested in. If index refers to a nonexistent element, then a runtime error occurs. If index is valid, then set assigns the elem parameter to the specified element. Note that elem is declared with E for its type. As with the set method, the E represents the data type of the ArrayList's elements. So elem is the same type as the type of ArrayList's elements. This example illustrates what we're talking about:

```
String mixedColor;
ArrayList<String> colors = new ArrayList<>();

colors.add("red");
colors.add("green");
colors.add("blue");
mixedColor = colors.get(0) + colors.get(1);
colors.set(2, mixedColor);
```

Note that mixedColor is declared to be a string and colors is declared to be an ArrayList of strings. So in the last statement, when we use mixedColor as the second argument in the set method call, the argument is indeed the same type as the type of color's elements.

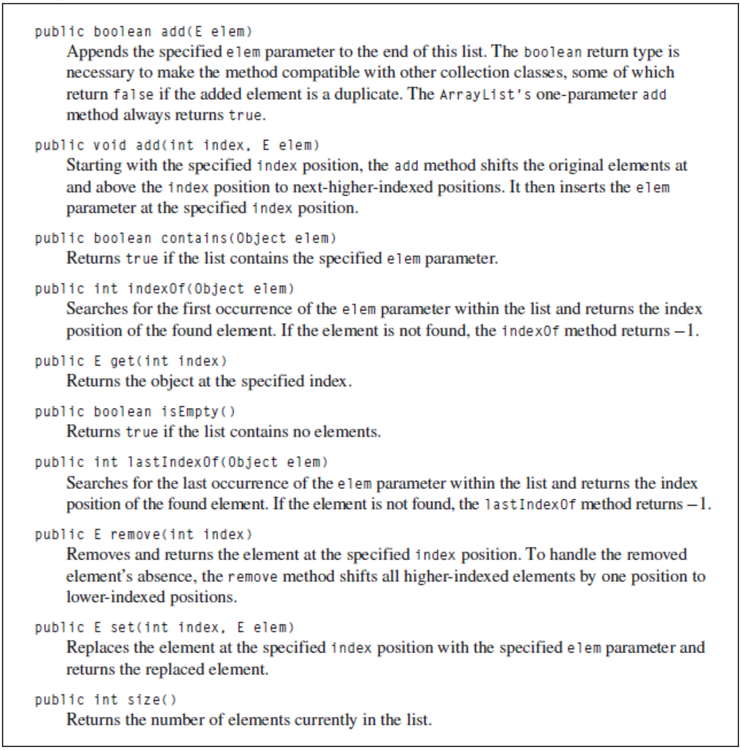
Can you determine what the colors ArrayList looks like after the code fragment executes? Draw a picture of the colors ArrayList on your own before proceeding. When you’re done, compare your answer to this:



In the set method’s API heading, note the return type, E. Most mutator/set methods simply assign a value and that’s it. In addition to assigning a value, the ArrayList’s set method also returns a value—the value of the specified element prior to the element being updated. Usually, there’s no need to do anything with the original value, so you just call set and the returned value dies. That’s what happens in the above code fragment. But if you want to do something with the original value, it’s easy to get it because set - returns it.

Additional ArrayList Methods

We've now explained the most important methods for the ArrayList class. There are quite a few more methods, and Figure 10.1 provides API headings and brief descriptions of the ones we have just described, as well as several others. As you read through the figure, we hope that you'll find most of the methods to be straightforward. But some items may need clarification. In searching an ArrayList for the first - occurrence of a passed-in elem parameter, the indexOf method declares elem's type to be Object. The Object type means the parameter may be any kind of object. Naturally, if the parameter's actual type is different from the type of elements in the ArrayList, then indexOf's search comes up empty and it returns −1 to indicate that elem was not found. By the way, we'll have lots more to say about the Object type (it's actually an Object class) in Chapter 14. Previously, we covered a one-parameter add method that adds an element at the end of the ArrayList. Figure 10.1's overloaded two-parameter add method adds an element at a specified position within the ArrayList.



`public boolean add(E elem)`
Appends the specified `elem` parameter to the end of this list. The `boolean` return type is necessary to make the method compatible with other collection classes, some of which return `false` if the added element is a duplicate. The `ArrayList's` one-parameter `add` method always returns `true`.

`public void add(int index, E elem)`
Starting with the specified `index` position, the `add` method shifts the original elements at and above the `index` position to next-higher-indexed positions. It then inserts the `elem` parameter at the specified `index` position.

`public boolean contains(Object elem)`
Returns `true` if the list contains the specified `elem` parameter.

`public int indexOf(Object elem)`
Searches for the first occurrence of the `elem` parameter within the list and returns the index position of the found element. If the element is not found, the `indexOf` method returns –1.

`public E get(int index)`
Returns the object at the specified index.

`public boolean isEmpty()`
Returns `true` if the list contains no elements.

`public int lastIndexOf(Object elem)`
Searches for the last occurrence of the `elem` parameter within the list and returns the index position of the found element. If the element is not found, the `lastIndexOf` method returns –1.

`public E remove(int index)`
Removes and returns the element at the specified `index` position. To handle the removed element's absence, the `remove` method shifts all higher-indexed elements by one position to lower-indexed positions.

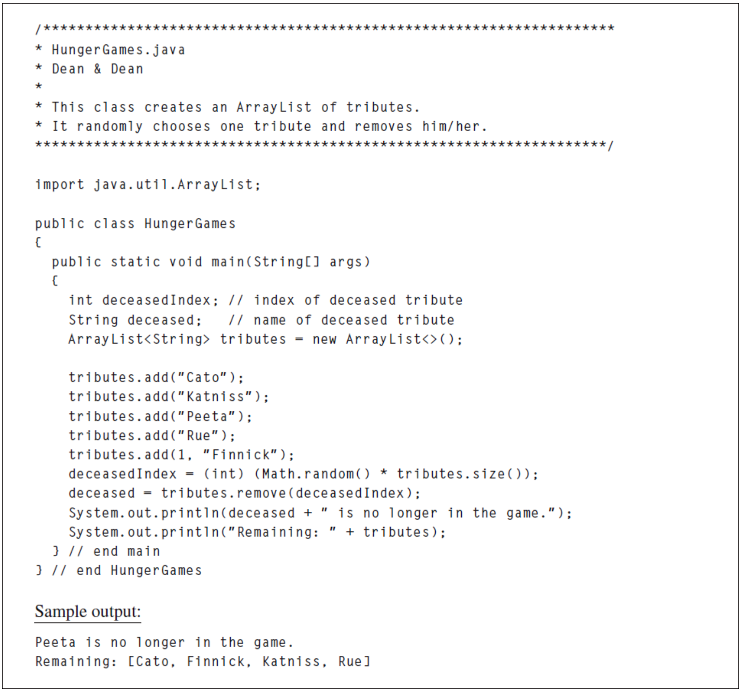
`public E set(int index, E elem)`
Replaces the element at the specified `index` position with the specified `elem` parameter and returns the replaced element.

`public int size()`
Returns the number of elements currently in the list.

Figure 10.1 API headings and descriptions for several ArrayList methods

Hunger Games Example

To reinforce what you've learned so far, let's take a look at how an ArrayList class is used in a complete working program. See the HungerGames program in Figure 10.2. It creates a list of tributes[1] by instantiating an ArrayList object and calling add to append tributes to the list. It then randomly chooses one of the tributes and removes that tribute from the list. It prints a message about the removed tribute and the remaining tributes.



```
/*********************************************************************
* HungerGames.java
* Dean & Dean
*
* This class creates an ArrayList of tributes.
* It randomly chooses one tribute and removes him/her.
*********************************************************************/

import java.util.ArrayList;

public class HungerGames
{
  public static void main(String[] args)
  {
    int deceasedIndex; // index of deceased tribute
    String deceased;   // name of deceased tribute
    ArrayList<String> tributes = new ArrayList<>();

    tributes.add("Cato");
    tributes.add("Katniss");
    tributes.add("Peeta");
    tributes.add("Rue");
    tributes.add(1, "Finnick");
    deceasedIndex = (int) (Math.random() * tributes.size());
    deceased = tributes.remove(deceasedIndex);
    System.out.println(deceased + " is no longer in the game.");
    System.out.println("Remaining: " + tributes);
  } // end main
} // end HungerGames
```

Sample output:

```
Peeta is no longer in the game.
Remaining: [Cato, Finnick, Katniss, Rue]
```

Figure 10.2 HungerGames program

Note the format of the tribute list in Figure 10.2's bottom output line—square brackets surrounding a comma-separated list. Can you find the source code that prints that list? If you're looking for square brackets and a loop, forget it—they're not there. So how in the world does the square-bracketed list get printed? In the final println statement at the bottom of the program, the tributes ArrayList gets concatenated to a string. That causes the JVM to do some work behind the scenes. If you attempt to concatenate an ArrayList to a string or print an ArrayList, the ArrayList returns a comma-separated list of ArrayList elements surrounded by square brackets ([]). And that's what happens when Figure 10.2's last statement executes.

10.3 Storing Primitives in an ArrayList

As mentioned earlier, ArrayLists store references. For example, in the Hunger Games, tributes is an ArrayList of strings, and strings are references. If you need to store primitives in an ArrayList, you can't do it directly, but if the primitives are wrapped up in wrapper classes,[2] you can store the resulting wrapped objects in an ArrayList. In this section, we show you how to do that.

Stock Average Example

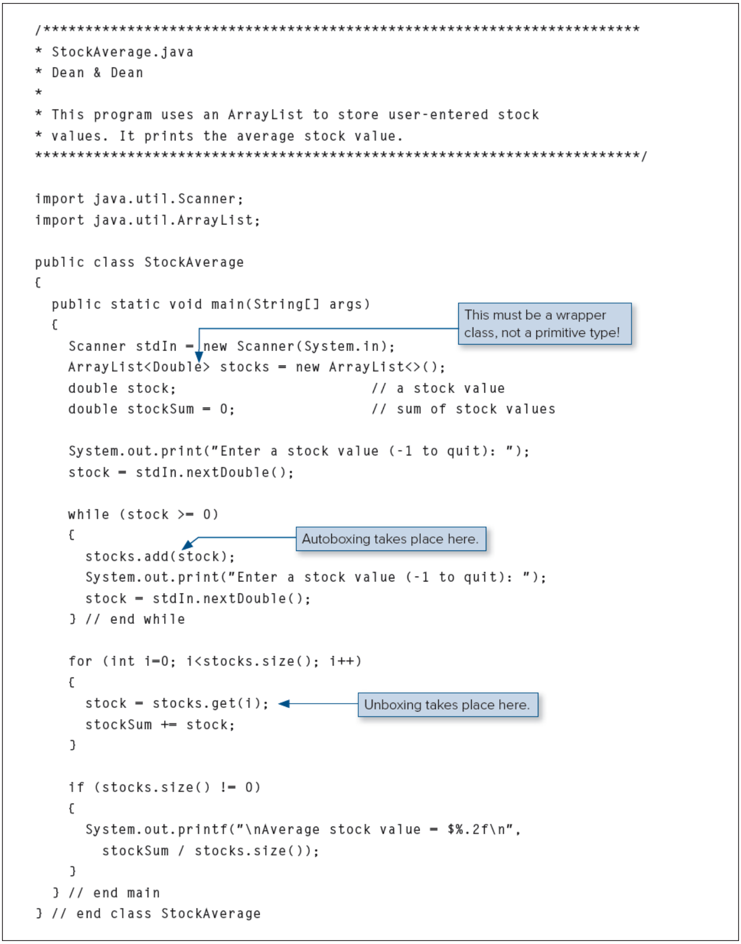
The StockAverage program in Figure 10.3 reads weighted stock values and stores them in an ArrayList. In simplified terms, a weighted stock value is the market price of one stock share times a number that scales that price up or down to reflect the importance of the stock's company in the overall marketplace. After the StockAverage program stores the weighted stock values in an ArrayList, the program calculates the average of all the entered weighted stock values. Why is an ArrayList appropriate for calculating a stock average? An ArrayList's size grows as necessary. That works well for stock averages because there are lots of stock averages (also called stock indexes), and they use different numbers of stocks in their calculations. For example, the Dow Jones Industrial Average uses stock values from 30 companies, while the - Russell 3000 Index uses stock values from 3,000 companies. Because it uses an ArrayList, the StockAverage program works well for both situations.

The StockAverage program stores stock values in an ArrayList named stocks. The stock values originate from user input in the form of doubles, like 25.6, 36.0, and so on. As you know, - ArrayLists can't store primitives; they can store references only. So the StockAverage program wraps up the doubles into Double wrapper objects just prior to storing them in the stocks ArrayList. As you might imagine, a *wrapper object* is an instance of a wrapper class, and each wrapper object stores one "wrapped up" primitive value. You don't have to worry very much about wrapper objects for ArrayLists. For the most part, you can pretend that ArrayLists can hold primitives. Case in point: The following line from the StockAverage program appears to add a primitive (stock) to the stocks ArrayList:

```
stocks.add(stock);
```

What actually happens behind the scenes is that the stock primitive gets converted automatically to a wrapper object prior to its being added to the stocks ArrayList. Really, there is just one thing you have to worry about when working with primitives in an ArrayList. When you create an ArrayList object to hold primitive values, the type you specify in the angled brackets must be the wrapped version of the primitive type, that is, Double instead of double, Integer instead of int, and so on. This line from the StockAverage program illustrates what we’re talking about:

Page 444



```
/***********************************************************************
* StockAverage.java
* Dean & Dean
*
* This program uses an ArrayList to store user-entered stock
* values. It prints the average stock value.
************************************************************************/

import java.util.Scanner;
import java.util.ArrayList;

public class StockAverage
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ArrayList<Double> stocks = new ArrayList<>();
    double stock;                     // a stock value
    double stockSum = 0;              // sum of stock values

    System.out.print("Enter a stock value (-1 to quit): ");
    stock = stdIn.nextDouble();

    while (stock >= 0)
    {
      stocks.add(stock);
      System.out.print("Enter a stock value (-1 to quit): ");
      stock = stdIn.nextDouble();
    } // end while

    for (int i=0; i<stocks.size(); i++)
    {
      stock = stocks.get(i);
      stockSum += stock;
    }

    if (stocks.size() != 0)
    {
      System.out.printf("\nAverage stock value = $%.2f\n",
        stockSum / stocks.size());
    }
  } // end main
} // end class StockAverage
```

This must be a wrapper class, not a primitive type!

Autoboxing takes place here.

Unboxing takes place here.

Figure 10.3 StockAverage program illustrating an ArrayList of Double objects

ArrayList<Double> stocks = new ArrayList<>();

Autoboxing and Unboxing

In most places, it’s legal to use primitive values and wrapper objects interchangeably. The way it works is that the JVM automatically wraps primitive values and unwraps wrapper objects when it’s appropriate to do so. For example, if the JVM sees an int value on the right of an assignment statement and an Integer variable at the left, it thinks to itself, hmmm, to make this work, I need to convert the int value to an Integer wrapper object. It then gets out its Styrofoam packing peanuts and duct tape and wraps up the int value into an Integer wrapper object. That process is called *autoboxing*. On the other hand, if the JVM sees an Integer wrapper object on the right of an assignment statement and an int variable at the left, it thinks to itself, hmmm, to make this work, I need to extract the int value from the Integer wrapper object. It then proceeds to tear off the Integer wrapper object's covering, and it gets the int value that's inside. That process is called *unboxing*.

More formally, autoboxing is the process of automatically wrapping a primitive value in an appropriate wrapper class whenever there's an attempt to use a primitive value in a place that expects a reference. Refer to the stocks.add(stock); statement in Figure 10.3. That statement causes autoboxing to occur. The stocks.add method call expects a reference argument. Specifically, it expects the argument to be a reference to a Double wrapper object (since stocks is declared to be an ArrayList of Double references). When the JVM sees a primitive value argument (stock), it automatically wraps the argument in a Double wrapper class.

More formally, unboxing is the process of automatically extracting a primitive value from a wrapper object whenever there's an attempt to use a wrapper object in a place that expects a primitive. Refer to the stock = stocks.get(i); statement in Figure 10.3. That statement causes unboxing to occur. Because stock is a primitive variable, the JVM expects a primitive value to be assigned to it. When the JVM sees a wrapper object on the right of the assignment statement (stocks holds Double wrapper objects and get(i) retrieves the ith such wrapper object), it automatically extracts the primitive value from the wrapper object.

Autoboxing and unboxing take place automatically behind the scenes. That makes the programmer's job easier. Yeah!

Explicit Conversion Between

Primitives and Wrapper Objects

To convert a primitive value into its corresponding object explicitly, use the static valueOf method with the primitive value for its argument. To use this technique in Figure 10.3's StockAverage program, you would replace the autoboxing statement (stocks.add(stock);) with this:

```
stocks.add(Double.valueOf(stock));
```

To go the other way and extract a double value from a Double object - explicitly, you can have a Double wrapper object call its doubleValue method. To use this technique in Figure 10.3's StockAverage program, you would replace the unboxing statement (stock = (stocks.get(i));) with this:

```
stock = stocks.get(i).doubleValue();
```

As mentioned earlier, stocks holds Double wrapper objects, and get(i) retrieves the ith such wrapper object. The doubleValue() method call then extracts the primitive double value from the retrieved wrapper object.

10.4 ArrayList Example Using Anonymous Objects and the For-Each Loop

---

Anonymous objects and *for-each loops* are programming constructs that are particularly useful when used in conjunction with ArrayLists. In this section, we present for-each loop details and anonymous object details by showing how they're used in the context of an ArrayList program. But before we get to the program, we provide brief introductions for the two new constructs.

Usually, when you create an object, you immediately store the object's reference in a reference variable. That way, you can refer to the object later on by using the reference variable's name. If you create an object and don't immediately assign the object's reference to a reference variable, you've created an anonymous object. It's called anonymous because it doesn't have a name.

A for-each loop is a modified version of the traditional for loop. It can be used whenever there's a need to iterate through all of the elements in a collection of data. An ArrayList is a collection of data, and, as such, for-each loops can be used to iterate through all of the elements in an ArrayList.

A Bear-Store Example

Suppose you want to model a store which sells customized toy bears. You need a Bear class to represent each bear, a BearStore class to represent the store, and a BearStoreDriver class to "drive" the program. Let's start by examining the Bear class in Figure 10.4. The Bear class defines two instance named constants which represent two permanent properties of a particular bear: (1) MAKER, the bear's manufacturer, such as Gund, and (2) TYPE, the bear's type, such as "pooh bear" or "angry campground bear." A constructor initializes these two instance constants, and a display method displays them.

Now let's examine the first part of the BearStore class, shown in Figure 10.5a. The BearStore class has one instance variable, bears, which is declared to be an ArrayList of Bear references. It holds the store's collection of toy bears. The BearStore class's addStdBears method fills the bears ArrayList with a specified number of standard teddy bears. Here's the statement that adds one standard teddy bear to the ArrayList:

```
bears.add(new Bear("Acme", "brown teddy"));
```

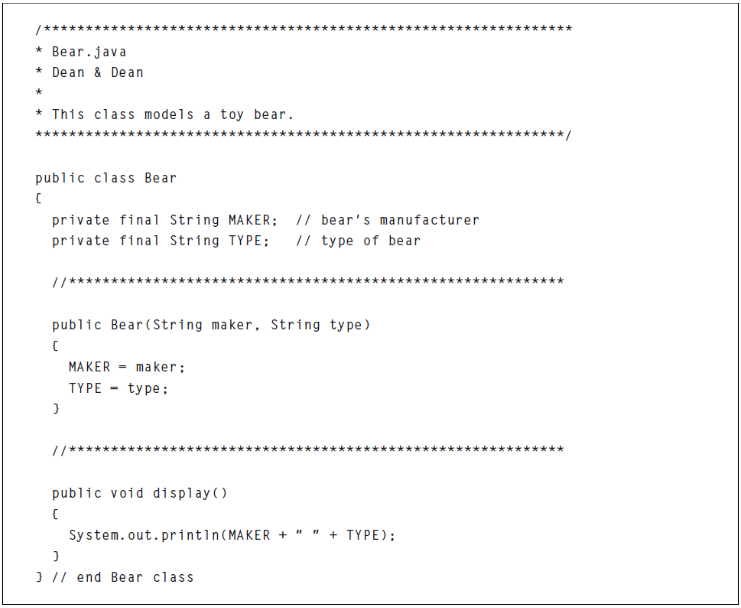
The statement instantiates a Bear object and passes the Bear object's reference to the bears.add method call. The statement does not assign the Bear object's reference to a Bear reference variable. Because there's no assignment to a Bear reference variable, that's an example of an anonymous object. As an alternative, the statement could have been written with a Bear reference variable like this:

```
Bear stdBear = new Bear("Acme", "brown teddy");
bears.add(stdBear);
```

But why bother with using two statements instead of one? The new bear's reference gets stored in the bears ArrayList and that's where it's processed. There's no need to store it in a second place (e.g., in the stdBear reference variable), so in the interest of code compactness, don't.

Now let's examine the bottom part of the BearStore class, shown in Figure 10.5b. The BearStore class's getUserSpecifiedBear method prompts the user for a customized bear's maker and type and returns the newly created bear. Here's the return statement:

return new Bear(maker, type);



```
/*******************************************************************
* Bear.java
* Dean & Dean
*
* This class models a toy bear.
*******************************************************************/

public class Bear
{
  private final String MAKER;  // bear's manufacturer
  private final String TYPE;   // type of bear

  //*************************************************************

  public Bear(String maker, String type)
  {
    MAKER = maker;
    TYPE = type;
  }

  //*************************************************************

  public void display()
  {
    System.out.println(MAKER + " " + TYPE);
  }
} // end Bear class
```

Figure 10.4 Class that represents a toy bear

Note that there's no reference variable for the new bear. Thus, the new bear is considered to be an anonymous object. The return statement returns the new bear to the addUserSpecifiedBears method, where it gets added to the bears ArrayList.

When to Use an Anonymous Object

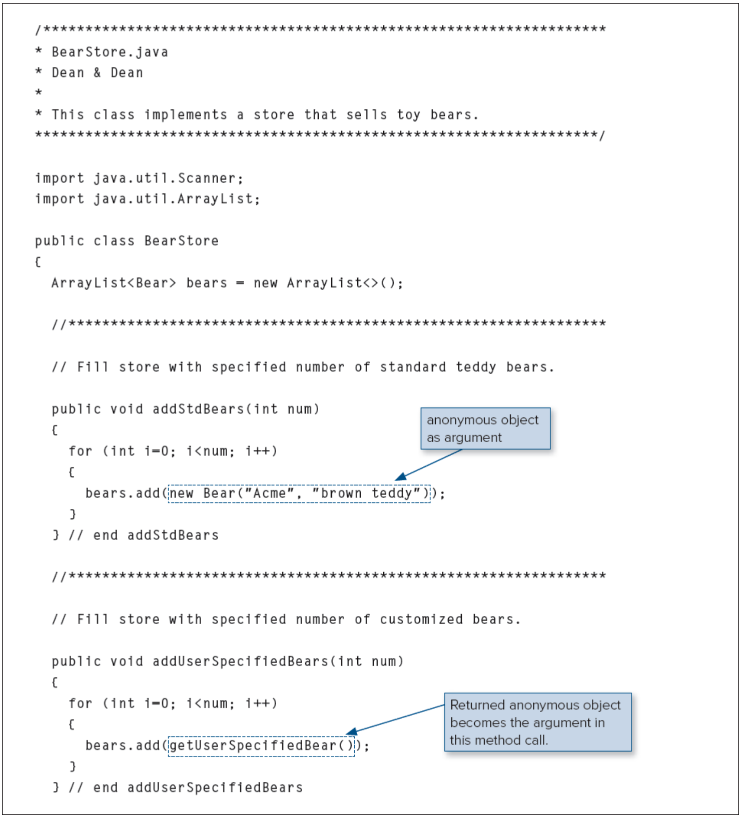
The bear-store program contains several specific examples of using anonymous objects. In general, you'll see anonymous objects being used in two circumstances:

1. When passing a newly created object into a method or constructor. For example:

   ```
   bears.add(new Bear("Gund", "Teddy"));
   ```

2. When returning a newly created object from a method. For example:

```
return new Bear(maker, type);
```



```
/*********************************************************************
* BearStore.java
* Dean & Dean
*
* This class implements a store that sells toy bears.
*********************************************************************/

import java.util.Scanner;
import java.util.ArrayList;

public class BearStore
{
  ArrayList<Bear> bears = new ArrayList<>();

  //*****************************************************************

  // Fill store with specified number of standard teddy bears.

  public void addStdBears(int num)
  {
    for (int i=0; i<num; i++)
    {
      bears.add(new Bear("Acme", "brown teddy"));
    }
  } // end addStdBears

  //*****************************************************************

  // Fill store with specified number of customized bears.

  public void addUserSpecifiedBears(int num)
  {
    for (int i=0; i<num; i++)
    {
      bears.add(getUserSpecifiedBear());
    }
  } // end addUserSpecifiedBears
```

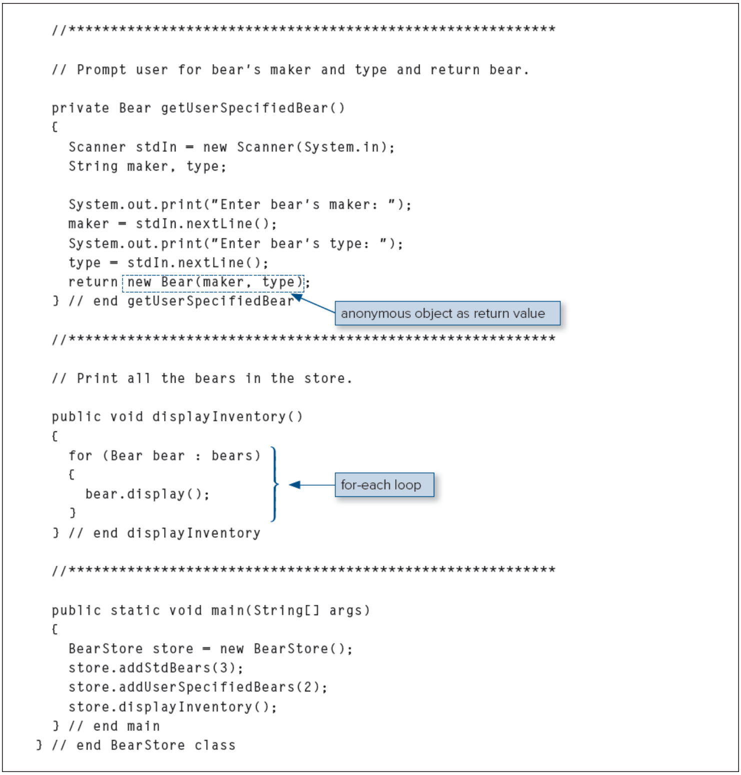
anonymous object as argument

Returned anonymous object becomes the argument in this method call.

Figure 10.5a Class that implements a toy-bear store—part A

Embedded Driver

At the bottom of the BearStore class, we've embedded the program's driver, main. It instantiates a BearStore object, adds three standard bears to the bear store, adds two user-specified bears to the bear store, and then displays the store's inventory of bears by calling displayInventory. In displaying the store's inventory, the displayInventory method accesses each bear in the bears ArrayList with the help of a for-each loop. In the next subsection, you'll learn about for-each loop details.



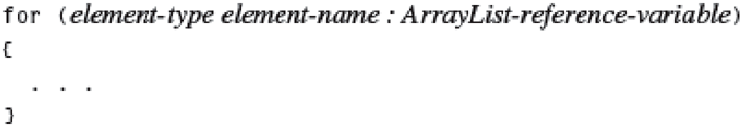
```
  //**********************************************************

  // Prompt user for bear's maker and type and return bear.

  private Bear getUserSpecifiedBear()
  {
    Scanner stdIn = new Scanner(System.in);
    String maker, type;

    System.out.print("Enter bear's maker: ");
    maker = stdIn.nextLine();
    System.out.print("Enter bear's type: ");
    type = stdIn.nextLine();
    return new Bear(maker, type);
  } // end getUserSpecifiedBear

  //**********************************************************

  // Print all the bears in the store.

  public void displayInventory()
  {
    for (Bear bear : bears)
    {
      bear.display();
    }
  } // end displayInventory

  //**********************************************************

  public static void main(String[] args)
  {
    BearStore store = new BearStore();
    store.addStdBears(3);
    store.addUserSpecifiedBears(2);
    store.displayInventory();
  } // end main
} // end BearStore class
```

Figure 10.5b Class that implements a

toy-bear store—part B

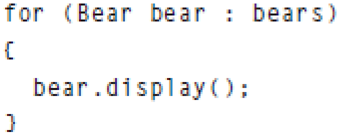
For-Each Loop

As mentioned earlier, you can use a for-each loop whenever you need to iterate through all the elements in a collection of data. Here is the for-each loop syntax for an ArrayList:



```
for (element-type element-name : ArrayList-reference-variable)
{
  . . .
}
```

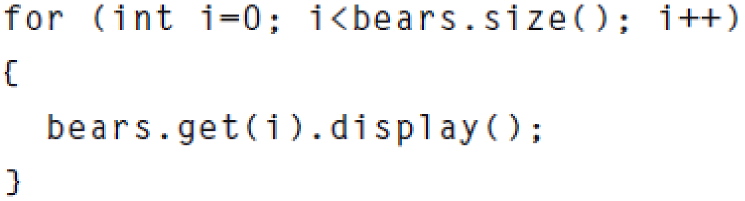
And here is an example for-each loop from Figure 10.5b’s displayInventory method:



```
for (Bear bear : bears)
{
  bear.display();
}
```

Note how the for-each loop header matches the preceding syntax: bears is an ArrayList reference variable, bear is the name of an element in the bears ArrayList, and Bear is the type for each element. It’s legal to choose any name for the element, and it’s common practice to use an abbreviation, but here we chose the descriptive name, bear. With each iteration of the for-each loop, you use the element’s name to refer to the current element. For example, bear.display() calls the display method for the current bear element.

Note that, as an alternative, you could implement the displayInventory method using a traditional for loop rather than a for-each loop. Here’s an implementation with a traditional for loop:

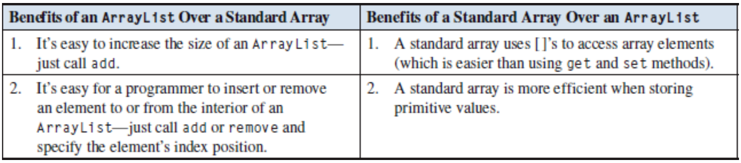


```
for (int i=0; i<bears.size(); i++)
{
  bears.get(i).display();
}
```

The for-each loop implementation is better because it is simpler. There’s no need to declare an index variable, and there’s no need to calculate and specify the ArrayList’s first and last index values.

10.5 ArrayLists Versus Standard Arrays

There's a lot of overlap in the functionality of an ArrayList and a standard array. So how can you tell which one to use? Your answer will be different for different situations. When deciding on an implementation, consider this table:



Benefits of an `ArrayList` Over a Standard Array	Benefits of a Standard Array Over an `ArrayList`
1. It's easy to increase the size of an `ArrayList`—just call `add`.	1. A standard array uses [ ]'s to access array elements (which is easier than using `get` and `set` methods).
2. It's easy for a programmer to insert or remove an element to or from the interior of an `ArrayList`—just call `add` or `remove` and specify the element's index position.	2. A standard array is more efficient when storing primitive values.

In looking at the table's first ArrayList benefit, easy to increase the size of an ArrayList, think about how much work is required to increase the size of a standard array. For a standard array, the programmer needs to instantiate a larger array and then copy the old array's contents to the new larger array. On the other hand, for an ArrayList, the programmer simply needs to call the add method. Note that behind the scenes, the JVM has to put forth some effort in implementing the add method, but the effort is kept to a minimum. ArrayLists are implemented with the help of an underlying standard array. Usually, the underlying array has a greater number of elements than the ArrayList, so adding another element to the ArrayList is easy—the JVM just borrows an unused element from the underlying array. As a programmer, you don't have to worry about or code those details; the "borrowing" takes place automatically.

The table's second ArrayList benefit, easy for a programmer to insert or remove an element to or from the interior of an ArrayList, is true, but just because it's easy for programmers doesn't mean it's easy for the JVM. Actually, the JVM has to do quite a bit of work when it adds or removes from the interior of an ArrayList. To insert an element, the JVM has to adjust its underlying array by shifting higher indexed elements to make room for the new element. And to remove an element, the JVM has to adjust its underlying array by shifting higher indexed elements to overlay the removed element.

10.6 The LinkedList Class

In the previous chapter and in the first part of this chapter, you learned about arrays and ArrayLists, which store ordered lists of related data of the same type. Now it's time to discuss another structure that stores an ordered list of related data of the same type—a *linked list*. Whereas arrays and ArrayLists store adjacent elements in adjacent memory locations, a linked list uses a chain of references to connect the linked list’s elements. Historically, a linked list’s elements have been called *nodes*. We’ll stick with the term *element* because that’s what you’re used to, and Oracle uses *element* consistently when referring to items in an array, an ArrayList, and a linked list.

In Java, linked lists are normally implemented using the LinkedList class from the API library. LinkedList objects form elements that are *doubly linked*, which means that each element contains two references—one that points to the element before it, and one that points to the element after it. Study Figure 10.6 to see what we’re talking about. The figure shows how elements can be added to and removed from a linked list.

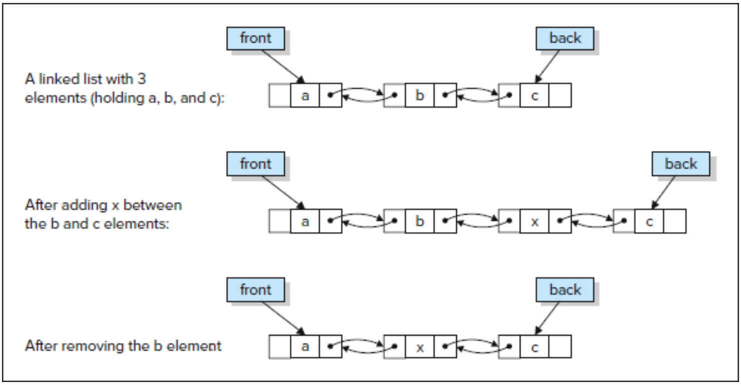

front
back
A linked list with 3 elements (holding a, b, and c):
a
b
c
front
back
After adding x between the b and c elements:
a
b
x
c
front
back
After removing the b element
a
x
c

Figure 10.6 Adding and removing elements in a doubly linked list

To create a linked list for storing the names of your favorite Android apps, instantiate the LinkedList class, like this:

```
LinkedList<String> androidApps = new LinkedList<>();
```

The name of the linked list, androidApps, is a reference variable that initially refers to null, which means that the linked list is empty.

A LinkedList implements many of the same methods an ArrayList implements. The add method is an important example. LinkedList's one-parameter add method appends an element to the end of the linked list, just like ArrayList's one-parameter add method appends an element to the high-index end of the ArrayList. The end of a linked list is normally called the list's *back*, and that's what we'll use, but it's sometimes also called the list's *tail*. LinkedList's two-parameter add method, where the first parameter is an index, can insert a new element anywhere in the linked list, just like ArrayList's two-parameter add method. The process of accessing a linked-list element starts from either of the two ends. Thus, it's easiest to find, add, or remove an element that's near either end. Once an element is found in the middle of a linked list, it's relatively easy to remove it or add a new one there, but if a linked list is large, it takes a relatively long time to find an element in the middle. That's much longer than it takes to find an element in the middle of an ArrayList, where the underlying array facilitates jumping quickly to any indexed location.

10.7 The List Interface

ArrayLists and LinkedLists implement many of the same methods—get, set, add, remove, clear, and size, to name a few. Using the same names is convenient because Java programmers have to remember only one set of names, not two. Furthermore, each of the same-named methods uses the same number of parameters and types of parameters. Once again, this makes it easy because Java programmers don’t have to remember as much. To encourage and standardize this strategy of having classes use methods with the same signatures (same method names and same parameter types), Java relies on *interfaces*. An interface is a template for designing classes that share certain qualities. The ArrayList and LinkedList classes both implement lists, and, as such, they share certain qualities common to all lists. Thus, to help with consistency, they are implemented using the List interface as a template. To implement a class using an interface, you just append an implements clause to the class's heading. For example, if you look up the ArrayList class in Java's API library, you'll see this:

public class ArrayList<E> implements List<E>

As indicated in Section 10.2, the E serves as a generic placeholder for the type specified when an ArrayList is declared in the future. For example, if you declare an ArrayList of strings with ArrayList<String> iPhoneApps;, then String will match up with E in ArrayList's class heading. Or, if you declare an ArrayList of Bear objects with ArrayList<Bear> bears;, then Bear will match up with E in ArrayList's class heading.

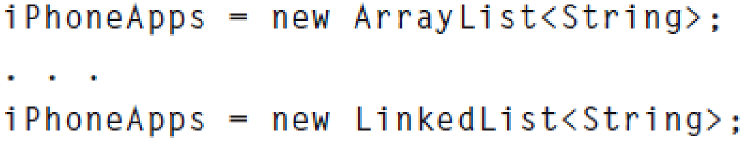
In the past, you've declared variables with a primitive type at the left or a class name at the left. For example:

```
double distance;
Student student;
```

You can also declare a variable with an interface at the left, like this:

```
List<String> iPhoneApps;
```

If you declare a variable as a double, then you are constrained to assigning numeric values to the variable. Likewise, if you declare a variable with an interface, you are constrained to assigning references to the variable such that the references point to instances of classes that implement the interface. For example, given the iPhoneApps declaration above, you can assign an ArrayList object to iPhoneApps and then, later, assign a LinkedList object to iPhoneApps. Here's the code:



```
iPhoneApps = new ArrayList<String>;
. . .
iPhoneApps = new LinkedList<String>;
```

This code works, but you might be thinking, "Why would I want to assign different type of lists into the same variable?" Suppose you want to write a method that reverses the order of the elements in a passed-in list parameter, and you want your method to be able to handle any type of list that implements the List interface. To do that, you should declare the list parameter with the List interface. Here's the appropriate method heading:

public void reverseList(List<String> list)

Because ArrayList and LinkedList both implement the List interface, you should be able to pass instances of either of those classes to the reverseList method, and the reverse process should work. Nice!

As you've seen, using an interface for a declaration type can make your code more flexible. On the other hand, using an interface for a declaration type also can constrain your code. It prevents variables from calling methods not specified by the interface. For example, LinkedList specifies several additional methods not specified in List. These additional LinkedList methods enable a LinkedList object to do things an ArrayList object cannot do. But if you declare the variable holding a LinkedList object to be of type List, that object cannot call any of these additional LinkedList methods.

10.8 Problem Solving: How to Compare Method

Execution Times

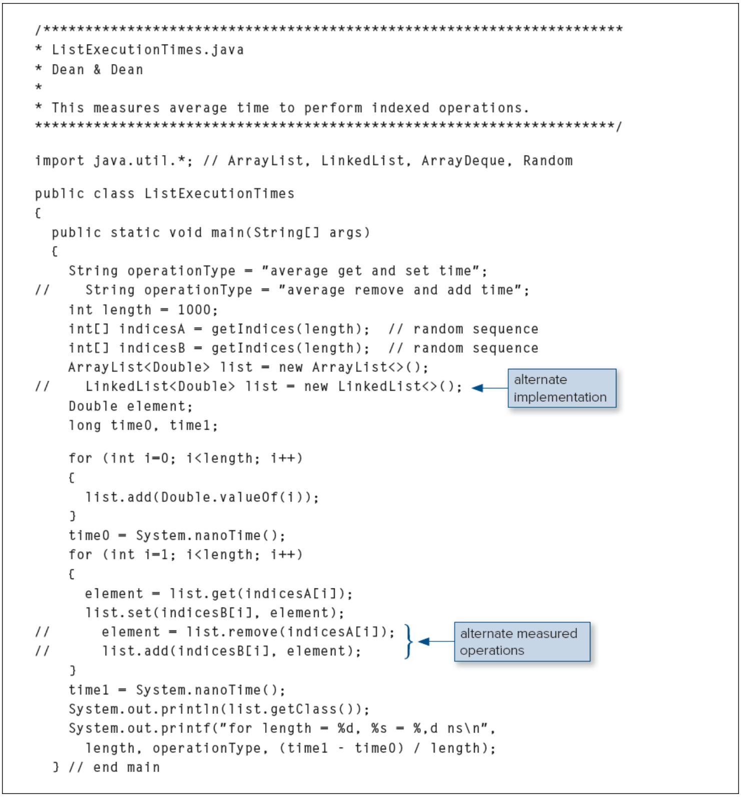
Because the LinkedList class and the ArrayList class both implement the List interface, an instance of either of these classes can call any of the methods specified in the List interface. Therefore, if you use only those methods specified by the List interface, you can freely substitute a LinkedList object for an ArrayList object, and vice versa. Thus, you could go back to any of the previous programs in this chapter, import the java.util package, and replace each occurrence of ArrayList with LinkedList. For example, in the BearStore program in Figure 10.5a, instead of ArrayList<Bear>, you could use LinkedList<Bear>. So if an ArrayList and a LinkedList produce the same results, how do you decide which to use? Use the one that runs faster. But how can you tell that? The best way to determine which of two implementations runs faster is to measure the times it takes them to do the same job. We could just tell you which type of list runs faster (and we'll do that eventually), but the main point of this section is to teach you how to determine the relative performance of methods, so you can apply the technique for other methods of interest.

To determine the execution time of a particular code fragment, surround that fragment with calls to System.nanoTime(), which returns current time in nanoseconds (billionths of a second). Of course, the measured time differences vary with hardware. Even on the same hardware, they vary from one run to another, due to unpredictable background activity. Still, comparative measurement is a useful technique because software performance differences are often quite large and easy to see.

For our first comparison, we'll look at the get and set operations, which are used to retrieve and update a specified list's element value, respectively. We'll determine how fast those operations execute when an ArrayList is used, and then when a LinkedList is used. We'll make those determinations with the help of the ListExecutionTimes program, shown in Figures 10.7a and 10.7b. In the first figure, note how the list variable is declared to be an ArrayList. Note also the commented-out code that declares the list variable as a LinkedList. We'll run the program twice, with the comment //'s switched the second time.

The ListExecutionTimes program measures the average time to retrieve each element at a random index and then update each element at a different random index. We arbitrarily fix the list length at 1,000, and call the getIndices helper method to generate two arrays of indices, indicesA and indicesB. Each of these arrays contains all the integers in the range between zero and length in a random sequence, and their sequences are different.

Page 455



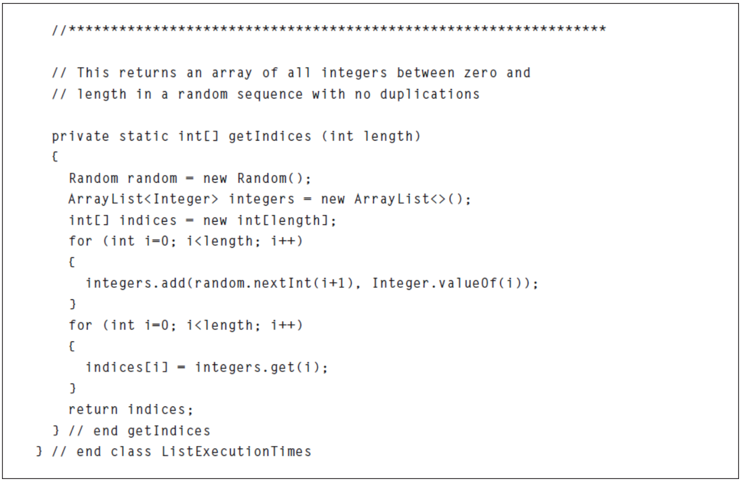
```
/*************************************************************************
* ListExecutionTimes.java
* Dean & Dean
*
* This measures average time to perform indexed operations.
*************************************************************************/

import java.util.*; // ArrayList, LinkedList, ArrayDeque, Random

public class ListExecutionTimes
{
  public static void main(String[] args)
  {
    String operationType = "average get and set time";
//    String operationType = "average remove and add time";
    int length = 1000;
    int[] indicesA = getIndices(length);  // random sequence
    int[] indicesB = getIndices(length);  // random sequence
    ArrayList<Double> list = new ArrayList<>();
//    LinkedList<Double> list = new LinkedList<>();
    Double element;
    long time0, time1;

    for (int i=0; i<length; i++)
    {
      list.add(Double.valueOf(i));
    }
    time0 = System.nanoTime();
    for (int i=1; i<length; i++)
    {
      element = list.get(indicesA[i]);
      list.set(indicesB[i], element);
//      element = list.remove(indicesA[i]);
//      list.add(indicesB[i], element);
    }
    time1 = System.nanoTime();
    System.out.println(list.getClass());
    System.out.printf("for length = %d, %s = %,d ns\n",
      length, operationType, (time1 - time0) / length);
  } // end main
```

alternate implementation

alternate measured operations

Figure 10.7a ListExecutionTimes program—part A For alternate evaluations, substitute commented

statements for preceding active statements.



```
  //*****************************************************************

  // This returns an array of all integers between zero and
  // length in a random sequence with no duplications

  private static int[] getIndices (int length)
  {
    Random random = new Random();
    ArrayList<Integer> integers = new ArrayList<>();
    int[] indices = new int[length];
    for (int i=0; i<length; i++)
    {
      integers.add(random.nextInt(i+1), Integer.valueOf(i));
    }
    for (int i=0; i<length; i++)
    {
      indices[i] = integers.get(i);
    }
    return indices;
  } // end getIndices
} // end class ListExecutionTimes
```

Figure 10.7b ListExecutionTimes program—part B This helper method generates a random sequence of all integers between zero and length.

In Figure 10.7a, note how calls to System.nanoTime() assign current times to time0 and time1. Because these assignments surround the loop that repeatedly calls get and set, (time1 - time0) calculates the total time it takes to perform all the get and set operations. At the bottom of Figure 10.7a, note the expression (time1 - time0) / length. That expression calculates the average time in nanoseconds for each pair of get and set method calls.

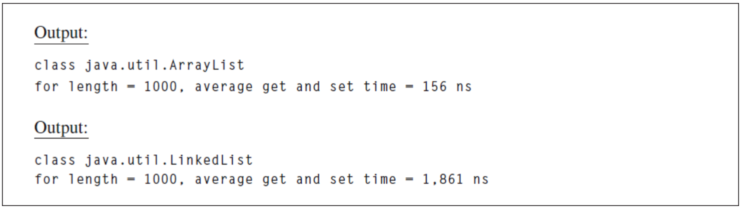
A representative execution time for get and set operations with an ArrayList appears in the first output session in Figure 10.8. To make the comparison with a LinkedList, in Figure 10.7a, we replace:

```
ArrayList<Double> list = new ArrayList<>();
```

with:

```
LinkedList<Double> list = new LinkedList<>();
```

Then we recompile and rerun to obtain the result shown in Figure 10.8's second output session. Although the measured times are sensitive to the hardware and background activity of the computer on which the program runs, their relative values should be reasonably accurate. The LinkedList implementation needs substantially more time to execute get and set because instead of jumping directly to the desired element, it must step through the list until the step count equals the index number. For length = 100, the average get and set time was about 401 nanoseconds (ns) for an ArrayList and about 1,248 ns for a LinkedList. For length = 10,000, the average get and set time was about 74 ns for an ArrayList and about 8,597 ns for a LinkedList. When the list is very long, a LinkedList is much slower.



Output:

```
class java.util.ArrayList
for length = 1000, average get and set time = 156 ns
```

Output:

```
class java.util.LinkedList
for length = 1000, average get and set time = 1,861 ns
```

Figure 10.8 Approximate average times to get and set an indexed element using the programs in Figures 10.7a and 10.7b. The first output uses an ArrayList version of the program. The second output uses a LinkedList version of the program.

For our second comparison, we look at two methods that alter the structure of a list—the remove method and the two-parameter add method. Traditionally, people expect a linked list to do better at operations that alter structure. Let's see if that's really the case. To compare the time to move an element from one index to another (by calling remove and add), in Figure 10.7a, we replace:

```
String operationType = "average get and set time";
```

with:

```
String operationType = "average remove and add time";
```

And we replace:

```
element = list.get(indicesA[i]);
list.set(indicesB[i], element);
```

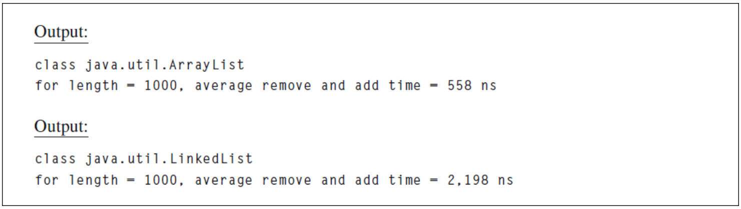
with:

```
element = list.remove(indicesA[i]);
list.add(indicesB[i], element);
```

Recompiling and rerunning with list implemented as an ArrayList generates Figure 10.9's first output session. Recompiling and rerunning with list implemented as a LinkedList generates Figure 10.9's second output session.

In executing the remove method, the JVM performs two operations—it first finds the element and then removes it. With an ArrayList, the find operation is fast (it uses the specified index to go directly to the element), but the remove operation is slow (after removing the element, it must repair its underlying array by shifting all higher elements down by one index number). A LinkedList has the opposite problem. With a LinkedList, the find operation is slow (because it must start its search from one of the ends), but the remove operation is fast (it just changes a couple of references).

Like the remove method, the two-parameter add method is also a two-step operation. First, it finds the element, and then it inserts the element in the list at the specified position. When choosing between an ArrayList and a LinkedList, the performance tradeoffs for add are the same as for remove.



Output:

```
class java.util.ArrayList
for length = 1000, average remove and add time = 558 ns
```

Output:

```
class java.util.LinkedList
for length = 1000, average remove and add time = 2,198 ns
```

Figure 10.9 Approximate average times to remove the element at one index and add it back at another index. The first output uses an ArrayList version of the program. The second output uses a LinkedList version of the program.

As Figure 10.9 shows, for most remove and add operations, an ArrayList's performance is better than a LinkedList's performance. This ArrayList advantage extends to very short lengths and very long lengths. For length = 100, the average remove and add time was about 655 ns for an ArrayList and about 1,288 ns for a LinkedList. For length = 10,000, the average move time was about 1,912 ns for an ArrayList and about 21,419 ns for a LinkedList.

Although these experiments show that an average ArrayList outperforms the corresponding LinkedList, a LinkedList outperforms an Arraylist for structural changes at the low-index end of the list. That’s because for an Arraylist, to add or remove an element at the low-index end of the list, the JVM must adjust the underlying array by shifting all higher elements up or down by one index position, respectively. This performance advantage for LinkedLists over ArrayLists is slight.

The relative performance of a LinkedList can improve when you add or remove a series of list elements that are near each other (directly connected or connected without too many intervening elements). The nearness is important because, as a programmer, you can use an *iterator* to keep track of the position of the most recently accessed element within a LinkedList. With iterators at your disposal, you don't need to start your search for an element at the ends. Instead, you can start your search at the most recently accessed element, and that can save time. To learn about iterators, look up ListIterator in Java's API library.

10.9 Queues, Stacks, and the ArrayDeque Class

To summarize what you've learned so far about collections of same-typed data: Use an array if you know in advance the number of elements, use an ArrayList if you need to add or remove elements anywhere in the list, and use a LinkedList in the special case where you need to add or remove a series of list elements that are near to each other. In this section, we present two other special cases—the case where you're allowed to add and remove elements only at the ends and the case where you're allowed to add and remove elements only at one end. Those two cases are handled by *queues* and *stacks*, respectively.

Queues

In society, a queue is a line of waiting people. New people are added to the back of the line as they arrive, and those who have waited the longest are removed from the front of the line as they are served. That strategy is called *first-in first-out (FIFO)* because the people who arrive first are served first. In the world of computer programming, a queue data structure mirrors a line of waiting people. With an ordinary queue, you're able to add elements only to the *back* end and remove elements only from the *front* end. Study Figure 10.10 to see what we're talking about.

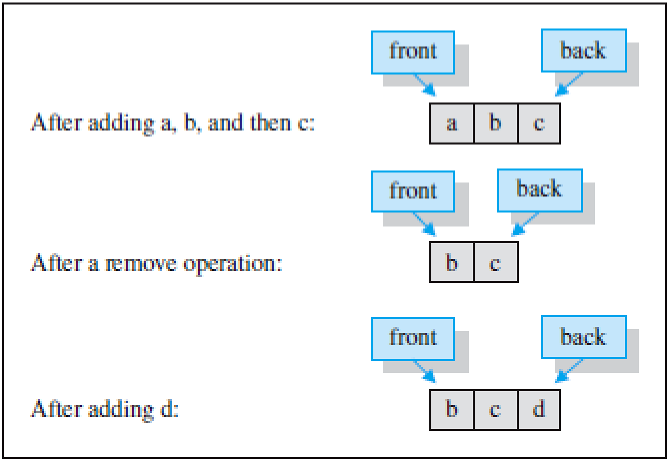

After adding a, b, and then c:
front
back
a
b
c
After a remove operation:
front
back
b
c
After adding d:
front
back
b
c
d

Figure 10.10 Queue operations

Although you can use an ArrayList or a LinkedList to implement a queue, for maximal efficiency, you should use the Java API ArrayDeque class. The *deque* (pronounced “deck”) in ArrayDeque stands for “double-ended queue.” With a double-ended queue, you can add or remove from either end.

Behind the scenes, an ArrayDeque is implemented with an array, which takes up less room in memory than a linked list, and that’s one of the reasons for its improved speed. By its nature, an array is a static entity, so to accommodate a queue's penchant for growing and shrinking dynamically, a little behind-the-scenes trickery is necessary. Trick #1: When an ArrayDeque is instantiated, the JVM creates a 16-element empty array. Anytime the array becomes full, and there's an attempt to add another element, the JVM replaces the array with a new array whose capacity is double that of the original array. Trick #2: Anytime there's an attempt to remove an element, there's no need to shift all of the higher-index elements to fill in the vacated front element's spot. Instead, the JVM adjusts the front of the queue, so it points to the next element after the original front element. This frees the original front element and makes it available to hold other elements at the back of the queue later on. Enabling a former front element to hold a future back element is accomplished by using a *circular array*. A circular array is built with a regular array. What makes it “circular” is its ability to treat the high-index element as adjacent to the 0-index element, so the elements form a continuous virtual circle. As part of an add operation, the circular array’s back variable will get incremented. If the incrementation causes the back variable’s value to become greater than the array’s highest index value, then the back variable gets reassigned to 0. The remove operation works the same, but it uses the front variable instead of the back variable.

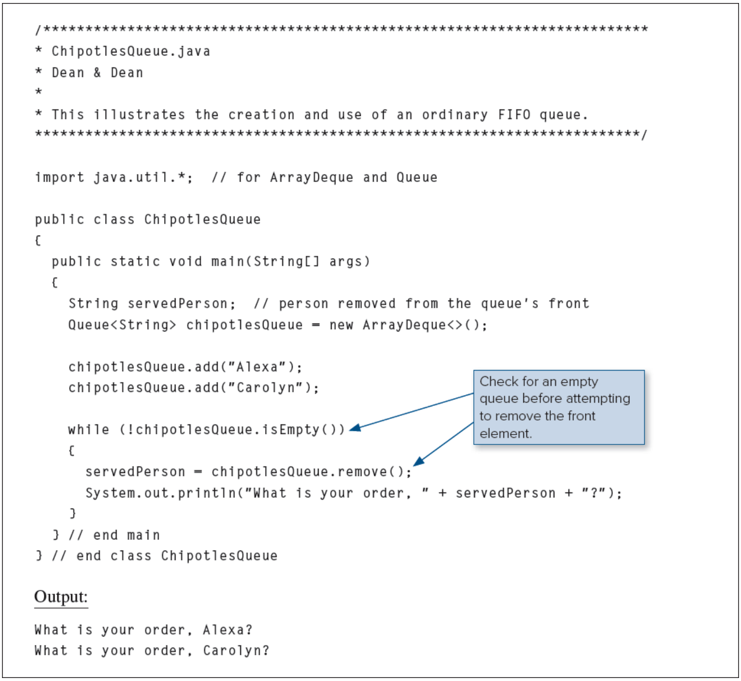
With the above strategy built into ArrayDeques, when you add an element to an ArrayDeque, the JVM simply assigns a value to an array element and updates the ArrayDeque’s internal back and size properties (except in the case when expanding the underlying array is necessary).

Removing an element from an ArrayDeque is similarly straightforward.

Now let’s dig into the details of how you can use ArrayDeques to create and manipulate queues in your own programs. To create an empty queue of people standing in line at a Chipotle Mexican Grill restaurant, instantiate an ArrayDeque like this:

```
Queue<String> chipotlesQueue = new ArrayDeque<>();
```

In the code above, note that chipotlesQueue is declared with Queue at its left. Queue is an interface. As explained previously, an interface can be used to constrain an object, so that the object can call only those methods that are included in the interface. The ArrayDeque class includes quite a few methods, and some of those methods are inappropriate for queues. For example, the ArrayDeque class includes a removeLast method, which removes the element at the back of the queue. Because ordinary queues are - supposed to remove elements only at the front, not at the back, the Queue interface does not include the removeLast method. Instead, it includes just the remove method, which attempts to remove the element at the front. Be aware that the remove method does not check the queue to see if it's empty before it attempts to remove the front element. So if you call remove from an empty queue, your program will crash. To prevent that, you can call Queue's isEmpty method before calling remove. To see what we're talking about, study the while loop in Figure 10.11's ChipotlesQueue program.



```
/************************************************************************
* ChipotlesQueue.java
* Dean & Dean
*
* This illustrates the creation and use of an ordinary FIFO queue.
************************************************************************/

import java.util.*;  // for ArrayDeque and Queue

public class ChipotlesQueue
{
  public static void main(String[] args)
  {
    String servedPerson;  // person removed from the queue's front
    Queue<String> chipotlesQueue = new ArrayDeque<>();

    chipotlesQueue.add("Alexa");
    chipotlesQueue.add("Carolyn");

    while (!chipotlesQueue.isEmpty())
    {
      servedPerson = chipotlesQueue.remove();
      System.out.println("What is your order, " + servedPerson + "?");
    }
  } // end main
} // end class ChipotlesQueue
```

Check for an empty queue before attempting to remove the front element.

Output:

```
What is your order, Alexa?
What is your order, Carolyn?
```

Figure 10.11 ChipotlesQueue program

Page 460

To add an element to the back of a queue, you can use Queue’s add method. For example, here’s how the ChipotlesQueue program adds “Alexa” to the food ordering line:

```
chipotlesQueue.add("Alexa");
```

Stacks

Like a queue, a *stack* is an ordered list that restricts element access. With a stack, you add and remove elements to and from only one end, and that end is called the *top*. A classic example is a stack of plates. If you're like most people, you arrange your plates in a stack and grab the top plate when you're ready to eat. After you clean and dry your plates, you place them back onto the top of the stack one at a time.[3] The stacking strategy is called *last-in first-out (LIFO)* because the last item placed on the top of the stack is the first item removed from the stack. Study Figure 10.12 to see how elements can be added to and removed from a stack. As indicated in the figure, for a stack, the add operation is called a *push*, and the remove operation is called a *pop*.

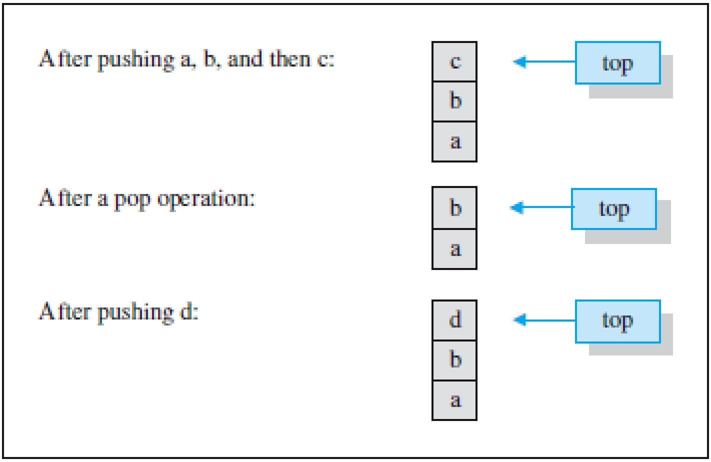
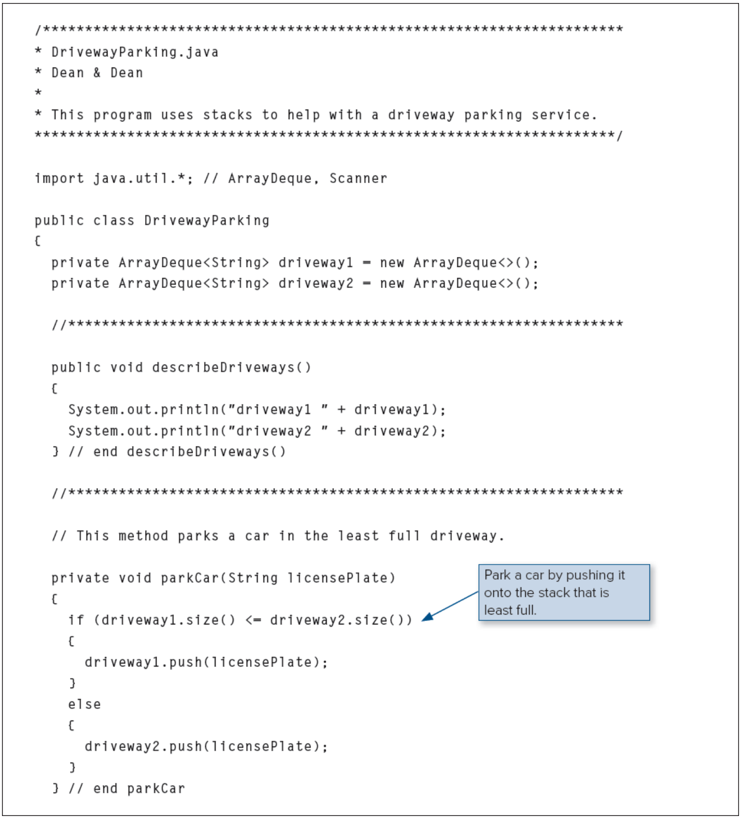

After pushing a, b, and then c:
c
b
a
top
After a pop operation:
b
a
top
After pushing d:
d
b
a
top

Figure 10.12 Stack operations

Stacks, like queues, can be implemented efficiently with ArrayDeques. For queues, there exists an interface, the Queue interface, that can be used to force an ArrayDeque to act like a queue. Unfortunately, there is no comparable interface to force an ArrayDeque to act like a stack. If you want to create a list that is constrained to stack operations (pushing to and popping from the stack's top), you can (1) use the Stack class instead of the ArrayDeque class, or (2) write your own stack class with an ArrayDeque instance variable and push and pop methods. There are many proponents of these two techniques, and you should feel free to use either of them. However, they both lead to relatively slow solutions (yes, the Java API Stack class is slower than the ArrayDeque class for stack operations). To avoid the performance hit, we won't worry about providing an airtight stack-operations-only constraint on the ArrayDeque class. We'll just use the ArrayDeque class as it should be used for a stack and assume that no one edits our source code later on by calling an inappropriate ArrayDeque method. (For example, calling removeLast would be inappropriate for a stack, because it removes the element at the bottom of the stack.)

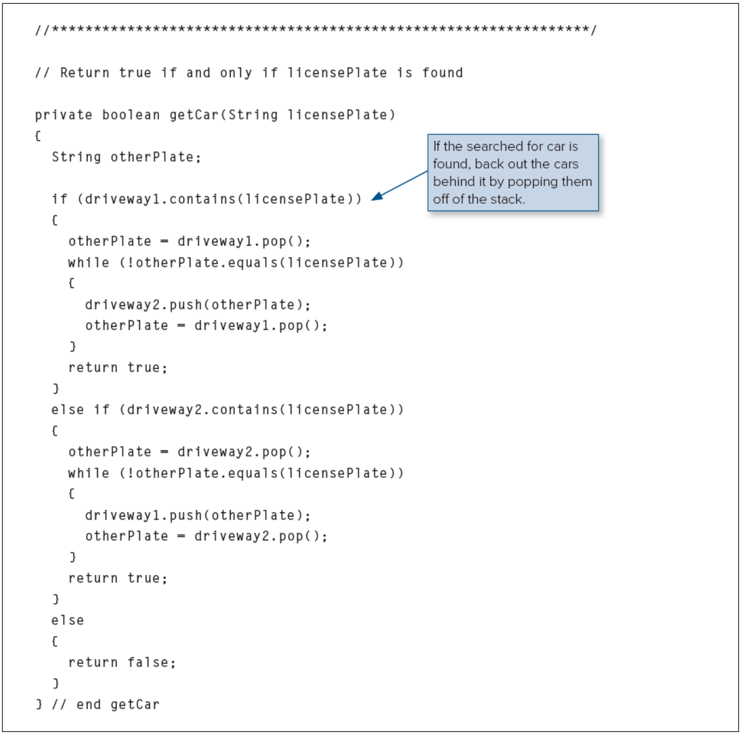
In Figures 10.13a, 10.13b, and 10.13c, we show a DrivewayParking program that illustrates how to use ArrayDeques for stacks. The program is designed to help with a low-budget parking service for college game-day events that parks cars in either of two long, narrow driveways. The program implements the driveways as ArrayDeque-based stacks. Here's the instantiation code:



```
/*************************************************************************
* DrivewayParking.java
* Dean & Dean
*
* This program uses stacks to help with a driveway parking service.
*************************************************************************/

import java.util.*; // ArrayDeque, Scanner

public class DrivewayParking
{
  private ArrayDeque<String> driveway1 = new ArrayDeque<>();
  private ArrayDeque<String> driveway2 = new ArrayDeque<>();

  //**********************************************************************

  public void describeDriveways()
  {
    System.out.println("driveway1 " + driveway1);
    System.out.println("driveway2 " + driveway2);
  } // end describeDriveways()

  //**********************************************************************

  // This method parks a car in the least full driveway.

  private void parkCar(String licensePlate)
  {
    if (driveway1.size() <= driveway2.size())
    {
      driveway1.push(licensePlate);
    }
    else
    {
      driveway2.push(licensePlate);
    }
  } // end parkCar
```

Park a car by pushing it onto the stack that is least full.

Figure 10.13a DrivewayParking program —part A



```
//***************************************************************/

// Return true if and only if licensePlate is found

private boolean getCar(String licensePlate)
{
  String otherPlate;

  if (driveway1.contains(licensePlate))
  {
    otherPlate = driveway1.pop();
    while (!otherPlate.equals(licensePlate))
    {
      driveway2.push(otherPlate);
      otherPlate = driveway1.pop();
    }
    return true;
  }
  else if (driveway2.contains(licensePlate))
  {
    otherPlate = driveway2.pop();
    while (!otherPlate.equals(licensePlate))
    {
      driveway1.push(otherPlate);
      otherPlate = driveway2.pop();
    }
    return true;
  }
  else
  {
    return false;
  }
} // end getCar
```

If the searched for car is found, back out the cars behind it by popping them off of the stack.

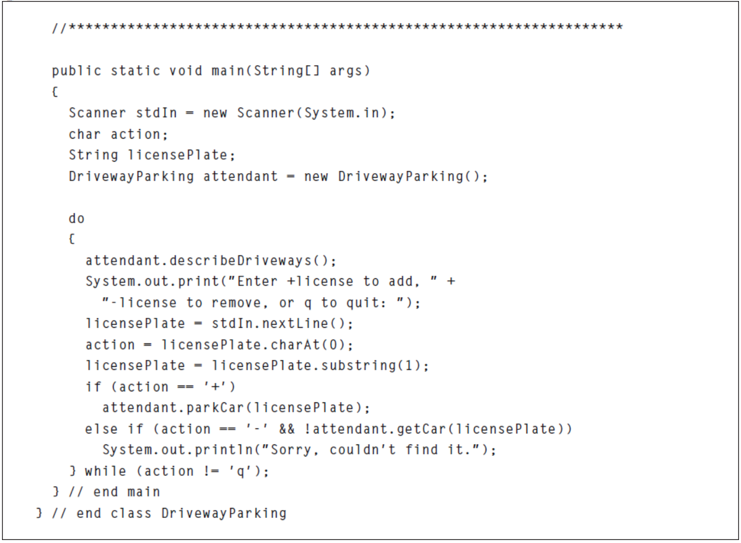
Figure 10.13b DrivewayParking program —part B

```
private ArrayDeque<String> driveway1 = new ArrayDeque<>();
private ArrayDeque<String> driveway2 = new ArrayDeque<>();
```

Page 462

Using stacks is helpful because their LIFO strategy mimics what happens as cars are parked and backed out of the driveways. When a customer arrives, the valet parks the customer’s car in the driveway that is less full and records the car’s license plate number by pushing it onto the less full stack. Here’s the relevant code from the DrivewayParking program’s parkCar method:

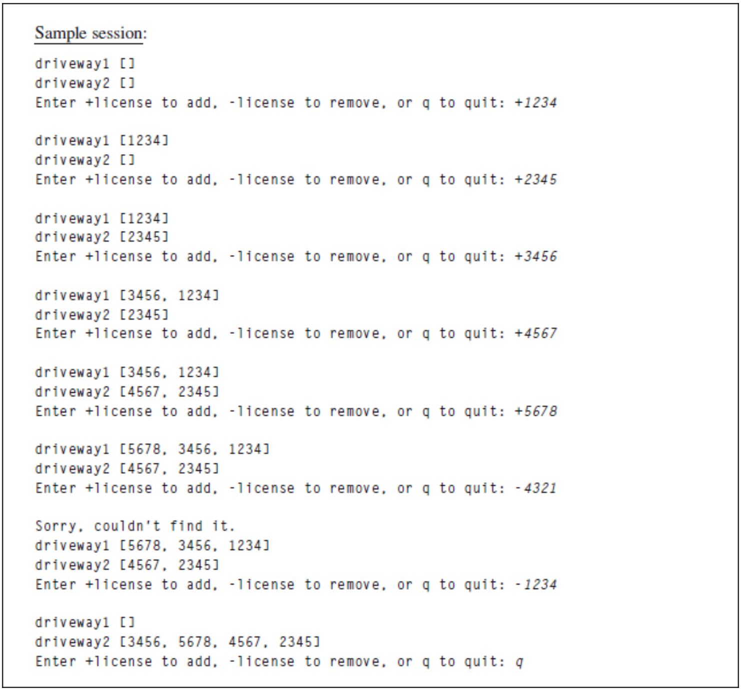
```
if (driveway1.size() <= driveway2.size())
{
  driveway1.push(licensePlate);
}
else
{
  driveway2.push(licensePlate);
}
```



```
    //*******************************************************************

    public static void main(String[] args)
    {
      Scanner stdIn = new Scanner(System.in);
      char action;
      String licensePlate;
      DrivewayParking attendant = new DrivewayParking();

      do
      {
        attendant.describeDriveways();
        System.out.print(”Enter +license to add, ” +
          ”-license to remove, or q to quit: ”);
        licensePlate = stdIn.nextLine();
        action = licensePlate.charAt(0);
        licensePlate = licensePlate.substring(1);
        if (action == '+')
          attendant.parkCar(licensePlate);
        else if (action == '-' && !attendant.getCar(licensePlate))
          System.out.println(”Sorry, couldn't find it.”);
      } while (action != 'q');
    } // end main
  } // end class DrivewayParking
```

Figure 10.13c DrivewayParking program —part C

When the customer returns, the valet determines where the car is by calling the ArrayDeque's contains method for each driveway stack. The contains method returns true if the passed-in license plate string is stored in the stack. If the license plate string is found in one of the stacks, cars are popped off of that stack (using the ArrayDeque's pop method) and pushed onto the other stack until the customer's car has been popped. The valet then mimics the program's popping and pushing by backing out each car that's behind the customer's car and parking it in the other driveway. To see how that works, mentally trace the DrivewayParking program, using Figure 10.14's sample session as a guide.



Sample session:

```
driveway1 []
driveway2 []
Enter +license to add, -license to remove, or q to quit: +1234

driveway1 [1234]
driveway2 []
Enter +license to add, -license to remove, or q to quit: +2345

driveway1 [1234]
driveway2 [2345]
Enter +license to add, -license to remove, or q to quit: +3456

driveway1 [3456, 1234]
driveway2 [2345]
Enter +license to add, -license to remove, or q to quit: +4567

driveway1 [3456, 1234]
driveway2 [4567, 2345]
Enter +license to add, -license to remove, or q to quit: +5678

driveway1 [5678, 3456, 1234]
driveway2 [4567, 2345]
Enter +license to add, -license to remove, or q to quit: -4321

Sorry, couldn't find it.
driveway1 [5678, 3456, 1234]
driveway2 [4567, 2345]
Enter +license to add, -license to remove, or q to quit: -1234

driveway1 []
driveway2 [3456, 5678, 4567, 2345]
Enter +license to add, -license to remove, or q to quit: q
```

Figure 10.14 DrivewayParking program sample session The driveway stacks are displayed with their tops at the left.

Page 464

10.10 Overview of The Java Collections Framework

The ArrayList, LinkedList, and ArrayDeque classes and the List, Queue, and Deque interfaces are a small part of a large coherent body of Java API software called the *Java collections framework.*[4] In addition to lists and queues, this framework includes interfaces and classes that describe and implement other types of data structures, like *sets* and *maps*.

Figure 10.15 shows part of the top of this framework. All of the interfaces and classes shown here are in the java.util package. The boxes in Figure 10.15 with italicized text are interfaces. The boxes with non-italicized text are instantiable classes. The Java collections framework has two interface hierarchies: the collection hierarchy, whose root is the Collection interface at the top of the diagram; and the map hierarchy, whose root is the Map interface, near the center of the diagram. The upward-pointing dashed arrows[5] from lower interfaces to upper interfaces point to more general interfaces, which the lower interfaces extend. *Extending* an interface means specifying additional methods. The upward-pointing dashed arrows from lower classes to upper interfaces point to interfaces which the lower classes implement. Those classes define all the methods specified by the interfaces they implement.

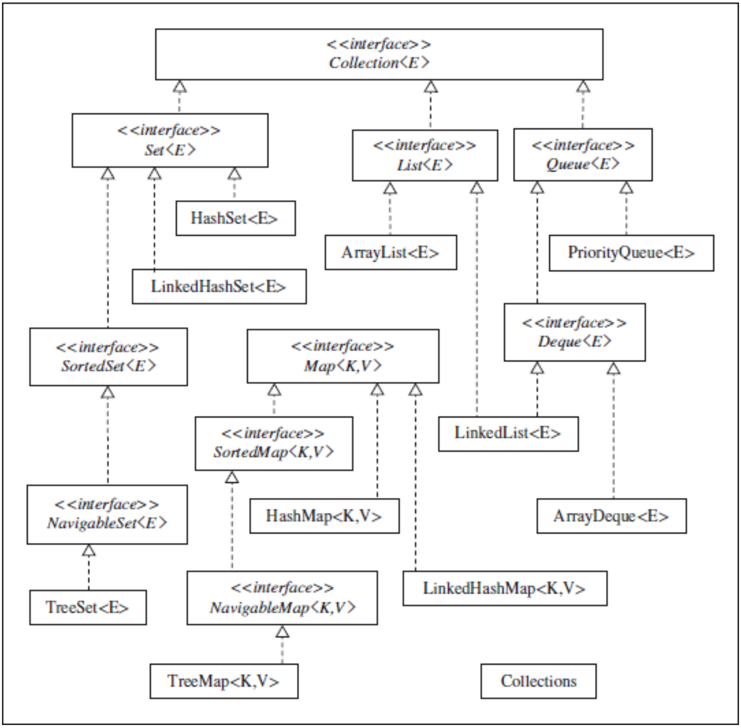

<<interface>>
Collection<E>
<<interface>>
Set<E>
<<interface>>
List<E>
<<interface>>
Queue<E>
HashSet<E>
ArrayList<E>
PriorityQueue<E>
LinkedHashSet<E>
<<interface>>
Deque<E>
<<interface>>
SortedSet<E>
<<interface>>
Map<K,V>
<<interface>>
SortedMap<K,V>
LinkedList<E>
<<interface>>
NavigableSet<E>
HashMap<K,V>
ArrayDeque<E>
TreeSet<E>
<<interface>>
NavigableMap<K,V>
LinkedHashMap<K,V>
TreeMap<K,V>
Collections

Figure 10.15 Part of the top of the Java

collections framework interface hierarchy

Now let’s reconsider the methods presented in Figure 10.1 in the context of Figure 10.15. Figure 10.15 shows that the ArrayList class implements the List interface. Oracle documentation says that the List interface specifies all the methods identified in Figure 10.1. The LinkedList class also implements the List interface. Thus, the ArrayList class and the LinkedList class both implement all the methods in Figure 10.1.

Oracle documentation also shows that the Collection interface specifies four of the methods in Figure 10.1: boolean add(E e), boolean contains(Object o), boolean isEmpty(), and int size(). Consequently, not just ArrayList and LinkedList, but more than fifty other classes that implement the Collection interface also implement these four methods in some way. Most of these other Collection classes do not appear in Figure 10.15, but Figure 10.15 does show a few of them: HashSet, LinkedHashSet, TreeSet, PriorityQueue, and ArrayDeque.

All classes in Figure 10.15 that implement the Collection interface provide not only zero-parameter constructors like those in this chapter's examples but also one-parameter constructors like these:

ArrayList(Collection<? extends E> c)

LinkedList(Collection<? extends E> c)

Such one-parameter constructors initialize the calling object, a Collection object, with the contents of a passed-in Collection object.

All classes that implement the Collection interface also provide an addAll method that adds the contents of any other collection object to the calling object:

public boolean addAll(Collection<? extends E> c)

If the calling object is an instance of a class that implements the List interface, the above addAll method appends the new elements to the end of the list.

As an alternative, that same List interface calling object can call this two-parameter addAll method:

public boolean addAll(int index, Collection<? extends E> c)

This inserts the passed-in collection elements into the calling object at the position indicated by the index parameter.

The List, Set, and Map interfaces themselves provide special methods, *static factory methods*, that use this syntax:

static <E> List<E> of(E... elements)

The E... elements parameter is a comma-separated list of objects. The of method returns an immutable instance of the List interface.

Because the List interface extends the Collection interface, instead of supplying a Collection argument to the preceding constructors and methods, you can supply a List argument generated by the above List.of method call. Although such a list is itself immutable, its contents become mutable when ingested by any of the preceding ArrayList or LinkedList constructors or addAll methods.

Thus, for example, in Figure 10.2's HungerGames program, instead of populating the ArrayList with separate add method calls, like this:

```
ArrayList<String> tributes = new ArrayList<>();
tributes.add(”Cato”);
tributes.add(”Katniss”);
tributes.add(”Peeta”);
tributes.add(”Rue”);
```

We could populate the ArrayList with one addAll method call, like this:

```
ArrayList<String> tributes = new ArrayList<>();
tributes.addAll(List.of(”Cato”, ”Katniss”, ”Peeta”, ”Rue”));
```

Or we could populate the ArrayList during its initialization like this:

```
ArrayList<String> tributes =
  new ArrayList<>(List.of(”Cato”, ”Katniss”, ”Peeta”, ”Rue”));
```

Page 467

Now let’s consider some of the other interfaces in Figure 10.15. You already know much of what the List interface specifies, and you know how useful it can be. The Set interface specifies just a few additional methods (like the of method) beyond those specified in the Collection interface. But it adds the stipulation: A Set contains no duplicate elements. This is a very useful property.

As mentioned earlier, the Queue interface specifies methods that describe waiting-line behavior. Its add method adds an element to the back of the queue. Its remove method removes an element from the front of the queue. Notice that the LinkedList class implements both List and Queue interfaces, and it also implements additional Deque methods, which allow additions to either end or removals from either end.

There's another type of queue, a priority queue, which behaves differently than a regular queue. A priority queue is a queue in which elements are ordered in priority order, regardless of the order in which they are added to the collection. A priority queue's add method inserts an element into the queue so that lower-valued elements are always removed before higher-valued elements. A priority queue's remove method removes the element with the lowest value. The Java collections framework implements a priority queue's functionality within its PriorityQueue class. The PriorityQueue class implements the Queue interface but not the Deque interface.

We put the Map interfaces in the middle of Figure 10.15 to emphasize that a map relates an object in a set (on the left side) to another object that could be either another set or a list or a queue (on the right side). A map can perform the same kind of mathematical operation a list can perform. Either of these structures works like a mathematical *function*. With a mathematical function, you supply an independent input value (like an ArrayList index), and the function gives back a corresponding output value (like the value at that index). A map generalizes this by also accepting non-integer inputs. For example, the independent variable in a map—the map's *key*—could be the wrapped form of an integer, a character, a string, a floating-point number, or any other type of object. The only constraint is that all keys must be distinct—no duplicates. Thus, a map's keys comprise a set that can be retrieved by calling the keySet method.

It's helpful to think of a map as a two-column lookup table. The first column contains items that you know (keys). The second column contains items that you want to find (values). To add a value of type V identified by a key of type K to a Java Map, use this method:

```
public V put(K key, V value)
```

If there is already a value associated with this key, the put method returns that pre-existing value. To retrieve a reference to the value identified by key, use this method:

```
public V get(Object key)
```

And to remove and return the value

identified by key, use this method:

```
public E remove(K key)
```

In the program in the next section, we'll use a map to relate the integer that serves as an object's identifier to the object it identifies. Whenever it can, that program will work with the identifying integer, but when it needs to access or modify a detail within the object, it will use the map to get the object and look at or change that internal detail.[6]

Note the Collections class in the bottom-right corner of Figure 10.15. It's a stand-alone utility class. It has no constructors, and all of its methods are static methods. These methods provide general services to other members of the Java collections framework. Some examples of useful Collections methods are addAll, binarySearch, fill, max, min, replaceAll, reverseOrder, rotate, shuffle, sort, and swap. You can learn more about these in the Java API documentation. The API method - headings in this class frequently employ strange-looking, generic-type syntax, but if you just do the most obvious thing, there's a good chance it will work and do what you want.

The program in the next section will use four of the classes in Figure 10.15—TreeSet (a set), LinkedHashSet (another set), ArrayDeque (a queue), and HashMap (a map). A TreeSet sorts its elements into a default or prescribed order. In the upcoming program, TreeSet elements will be integers, and by default, the TreeSet will keep them ordered from lowest to highest. A LinkedHashSet keeps its elements in the order of entry. The upcoming program will use an ArrayDeque to implement a FIFO queue. A TreeMap has the same order as its TreeSet keys. A LinkedHashMap has the same order as its HashSet keys. A HashMap's order is not predictable, but a HashMap is the most efficient type of map. Because order does not matter for a simple lookup, the upcoming program will employ a HashMap for its lookup table.

10.11 Collections Example—Information Flow in a Network of Friends

Now let's consider an example that illustrates use of the software in the Java collections framework. Here's the situation. Suppose there is a set of citizens, and each citizen has a random set of friends. Assume all friendships are mutual. That is, all the friends of any particular citizen consider that citizen to be one of their friends, too.

Different citizens typically have different friends and different numbers of friends.

After we create citizens and a friendship network, we look to see how information flows through the network by performing the following experiment: We select one particular citizen and give that citizen a message. We ask that citizen to send that message to all of his or her friends and ask those friends to pass it on to their friends. This continues until everyone who can be reached becomes informed.

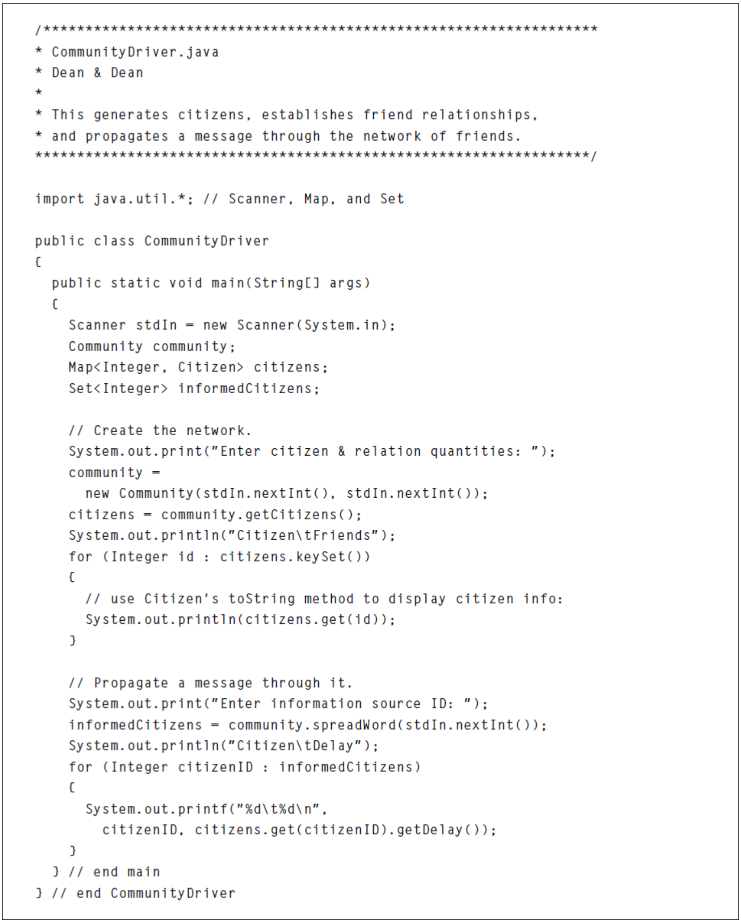
At the highest level, there are just two basic steps—build the network and distribute a message. So the top-level algorithm is easy. The devil is in the details. The most difficult aspects of building the network are deciding how to store the configuration information and deciding where to put the methods that assemble that information. The most difficult aspect of distributing the message is avoiding infinite repetition.

First, consider how to store configuration information. It’s difficult to draw a complete picture of a large random network, and it would be hard to store such a picture if it could be drawn. A better strategy is to store a list of the parts. We could store a list of citizens with associated relationships or we could store a list of relationships with associated citizens. It’s more intuitive to store the concrete items (the citizens) and associate the relationships with them. So that’s what we’ll do.

Because this program makes heavy use of sets, each citizen needs a unique identifier. The obvious way to initialize a citizen’s identifier is through a constructor parameter. But what if code calling that constructor happens to use the same ID in two different instantiations? How would we check for that possibility, and how would we respond if it occurred? We’ll avoid these problems by encapsulating ID assignment within the class that defines a citizen.

Top-Level Code

Figure 10.16 shows the program’s driver. In the main method, the first local variable is a Scanner object that retrieves keyboard input. The second local variable is an object that represents the community. The third local variable is a map of the community’s citizens. The fourth local variable is a set that eventually holds the results of a message-propagation experiment.



```
/*******************************************************************
* CommunityDriver.java
* Dean & Dean
*
* This generates citizens, establishes friend relationships,
* and propagates a message through the network of friends.
*******************************************************************/

import java.util.*; // Scanner, Map, and Set

public class CommunityDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Community community;
    Map<Integer, Citizen> citizens;
    Set<Integer> informedCitizens;

    // Create the network.
    System.out.print(”Enter citizen & relation quantities: ”);
    community =
      new Community(stdIn.nextInt(), stdIn.nextInt());
    citizens = community.getCitizens();
    System.out.println(”Citizen\tFriends”);
    for (Integer id : citizens.keySet())
    {
      // use Citizen’s toString method to display citizen info:
      System.out.println(citizens.get(id));
    }

    // Propagate a message through it.
    System.out.print(”Enter information source ID: ”);
    informedCitizens = community.spreadWord(stdIn.nextInt());
    System.out.println(”Citizen\tDelay”);
    for (Integer citizenID : informedCitizens)
    {
      System.out.printf(”%d\t%d\n”,
        citizenID, citizens.get(citizenID).getDelay());
    }
  } // end main
} // end CommunityDriver
```

Figure 10.16 CommunityDriver class

The first block of code creates the network. It asks the user to enter two integers separated by whitespace. The first entry is the total number of citizens. The second entry is the total number of distinct citizen-to-citizen relations (friendships). The next statement instantiates a Community object, whose constructor builds the network. The next statement gets a reference the citizen map. The for loop in this first block of code displays the community's citizens and their friends, with one citizen per row. In each row, the first column contains a citizen's ID, and the second column contains the IDs of all of that citizen's friends. The println statement in this for loop displays citizen information by using the Map called citizens to retrieve a particular citizen object and then asking that object to call its toString method.

The second block of code propagates a message through the network. It asks the user to enter the ID of the citizen who originates the message. The next statement asks the Community method called spreadWord to propagate the message. The remaining statements in this code block describe the message propagation. In each row, the first column contains the ID of a citizen receiving the message, and the second column contains the total number of steps required for the message to get to that citizen—the delay.

Network Composition

Figure 10.17a shows the first part of the Community class. The instance variable, citizens, refers to the map that contains citizen data. This map's keys are individual-citizen IDs, and the corresponding values are the individual-citizen objects corresponding to those IDs. Because the individual-citizen objects eventually include all friend relationships, this map is the repository for all network data. As you saw in the preceding driver code, it's also a handy tool for finding that data.

The Community constructor has

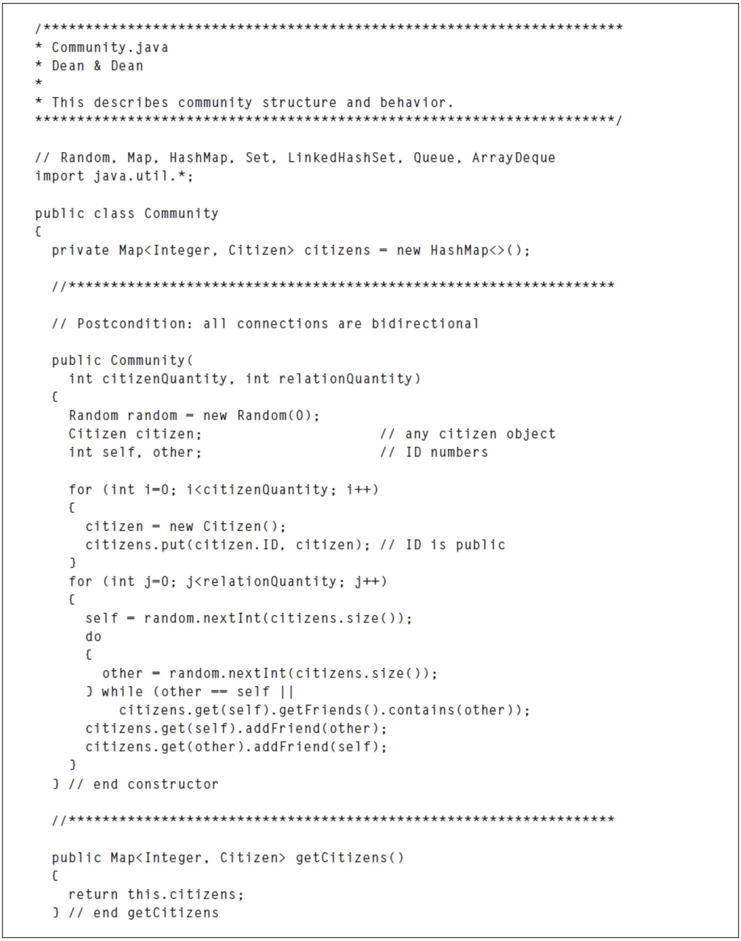
parameters for the total number of citizens and the total number of citizen-to-citizen interconnections. It creates a local variable for a random number generator called random. It declares a local variable for a Citizen object. It declares local variables for two citizen ID numbers. The first for loop instantiates each citizen, and using that citizen's internally generated ID, it enters that citizen into the general data repository—the map called citizens. This code assumes that the immutable citizen ID is a public instance variable.

The second for loop creates friend relationships. It randomly picks one citizen. Then it randomly picks another citizen who is not the first citizen and is not one of the first citizen's already-existing friends. Notice how it uses the Java Collection's contains method in the while condition of a do loop to perform expeditiously the tricky logic required to avoid duplication. After identifying a unique new relationship, this method adds each citizen involved in that relationship to the other citizen's set of friends. The pair of -addFriend method calls in the last two statements creates bidirectional links. For one-way links, you could omit one of these two method calls.

The getCitizens method at the bottom of Figure 10.17a would enable external methods to access the map called citizens. Once they have this map, they can use a simple citizen ID argument in a citizens.get method call to retrieve the corresponding Citizen object. You saw examples of these operations in the for loops in the driver in Figure 10.16.

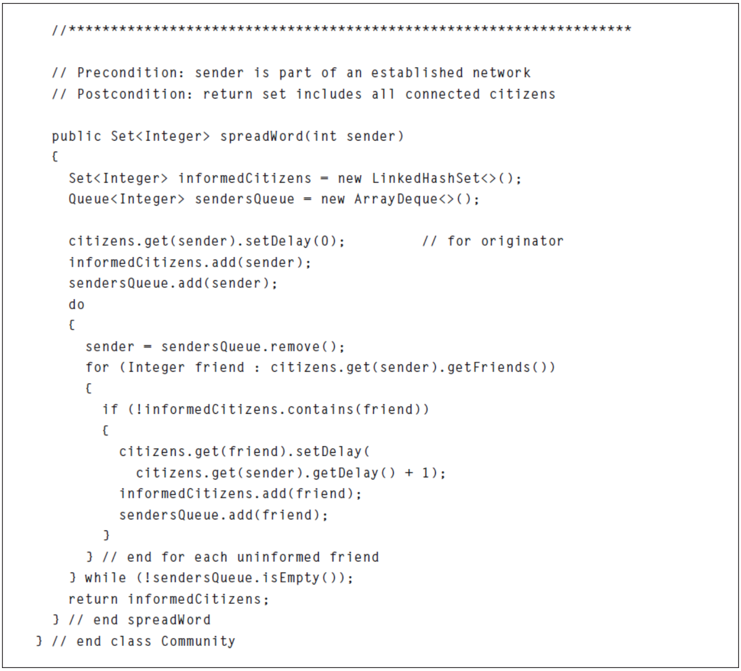
Message Propagation

Figure 10.17b contains the method that propagates a message through the network. This method's parameter, sender, initially contains the ID of the message originator. The Set called informedCitizens accumulates IDs of all citizens who have received the message. Because it's a LinkedHashSet, it keeps its contents in their order of entry. The Queue called sendersQueue is a FIFO queue that contains informed citizens who have not yet forwarded the message to their friends.



```
/*********************************************************************
* Community.java
* Dean & Dean
*
* This describes community structure and behavior.
*********************************************************************/

// Random, Map, HashMap, Set, LinkedHashSet, Queue, ArrayDeque
import java.util.*;

public class Community
{
  private Map<Integer, Citizen> citizens = new HashMap<>();

  //*****************************************************************

  // Postcondition: all connections are bidirectional

  public Community(
    int citizenQuantity, int relationQuantity)
  {
    Random random = new Random(0);
    Citizen citizen;                     // any citizen object
    int self, other;                     // ID numbers

    for (int i=0; i<citizenQuantity; i++)
    {
      citizen = new Citizen();
      citizens.put(citizen.ID, citizen); // ID is public
    }
    for (int j=0; j<relationQuantity; j++)
    {
      self = random.nextInt(citizens.size());
      do
      {
        other = random.nextInt(citizens.size());
      } while (other == self ||
          citizens.get(self).getFriends().contains(other));
      citizens.get(self).addFriend(other);
      citizens.get(other).addFriend(self);
    }
  } // end constructor

  //*****************************************************************

  public Map<Integer, Citizen> getCitizens()
  {
    return this.citizens;
  } // end getCitizens
```

Figure 10.17a Community class—part A This creates citizens and builds the network of friends.



```
  //*******************************************************************

  // Precondition: sender is part of an established network
  // Postcondition: return set includes all connected citizens

  public Set<Integer> spreadWord(int sender)
  {
    Set<Integer> informedCitizens = new LinkedHashSet<>();
    Queue<Integer> sendersQueue = new ArrayDeque<>();

    citizens.get(sender).setDelay(0);         // for originator
    informedCitizens.add(sender);
    sendersQueue.add(sender);
    do
    {
      sender = sendersQueue.remove();
      for (Integer friend : citizens.get(sender).getFriends())
      {
        if (!informedCitizens.contains(friend))
        {
          citizens.get(friend).setDelay(
            citizens.get(sender).getDelay() + 1);
          informedCitizens.add(friend);
          sendersQueue.add(friend);
        }
      } // end for each uninformed friend
    } while (!sendersQueue.isEmpty());
    return informedCitizens;
  } // end spreadWord
} // end class Community
```

Figure 10.17b Community class—part B This propagates a message through the network.

The setDelay method call records the relative time (delay) when a sender received the message he or she is forwarding. After setting the message originator's delay to zero, spreadWord adds the message originator's ID to informedCitizens and sendersQueue.

Then the spreadWord method enters

a do loop. The first statement in this do loop extracts the ID at the front of the queue and assigns it to sender. Initially, this ID extraction from the queue is just a re-assignment of the originator's ID. But in subsequent do-loop iterations, sender acts like a local variable and takes on different citizens' IDs.

With each new sender, spreadWord enters a for-each loop that loops through all of that sender's friends. If a particular friend is uninformed, that friend gets a delay equal to the sender's delay plus 1. Then that friend's ID goes into informedCitizens and sendersQueue. As this process proceeds, more and more friends become informed, and because informed citizens do not satisfy the if condition, they do not reenter the queue. With fewer and fewer new entries, eventually the queue becomes empty, the do loop terminates, and spreadWord returns the set of all informed citizens.

The Citizen Class

Figure 10.18 shows the Citizen class. The first statement defines a private static variable, nextID, which provides a unique identifier for each new instance. The second statement initializes the ID instance constant with the current value of nextID and then immediately increments nextID in preparation for the next instantiation. Because ID is a constant, we can make it public and avoid the definition and the calling of a getID method.

The friends instance variable is a reference to a set of friend IDs. We give this variable an instance of a TreeSet because we want the friends to be stored in numerical order. If the IDs were names instead of numbers, the variable declaration would be

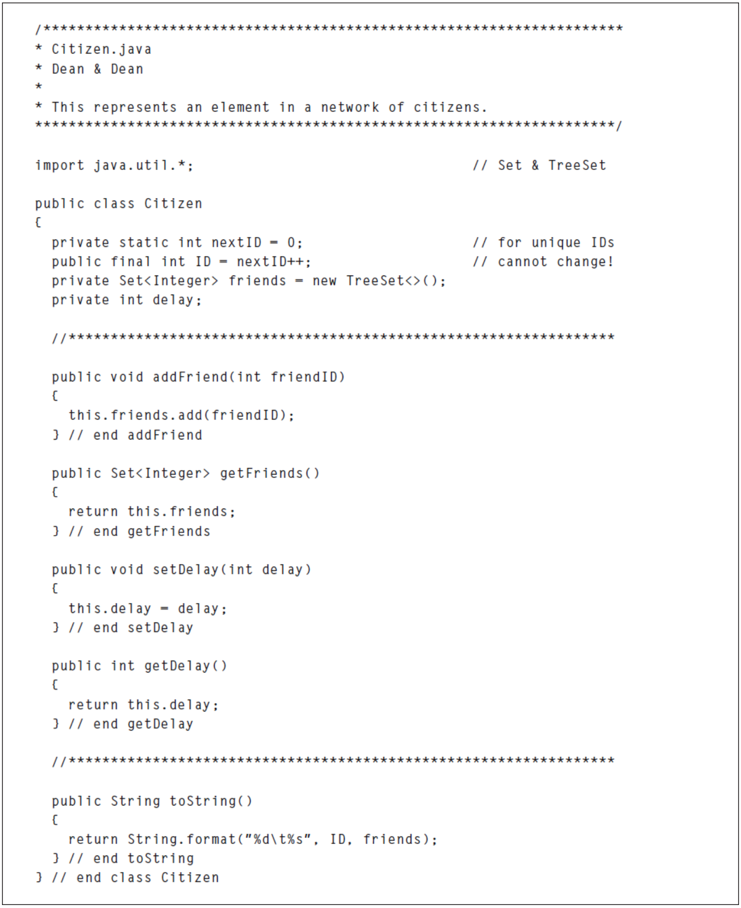
Set<String>, and the corresponding TreeSet would store friends in alphabetical order. If we had instantiated friends as a LinkedHashSet, this would store oldest friends first. The next instance variable, delay, records the relative time when the current citizen receives a message, and that depends on who starts the message.

Notice that there is no explicit constructor and there is no setID method. So it is impossible for an outsider to duplicate citizen identifiers accidentally. Citizen instantiation initializes the current citizen's ID, and it creates a set to hold friend IDs, but it does not populate that friends set. Making friends comes later, in separate successive calls to the addFriend method. The getFriends, setDelay, and getDelay methods are self-explanatory. The toString method displays the current citizen's ID. Then it displays the IDs of that citizen's friends, using the default toString method provided by the Java API TreeSet class.

Typical Output

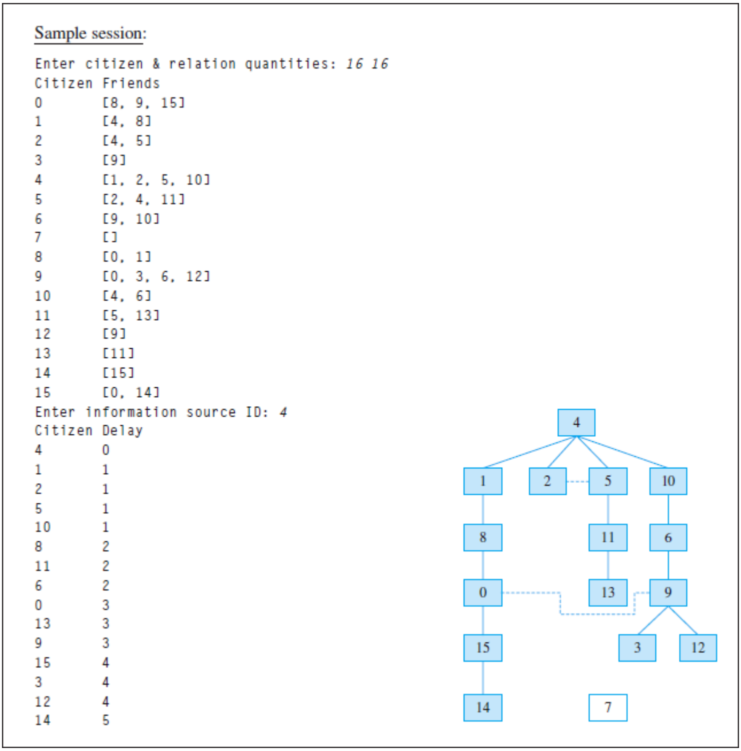
Figure 10.19 shows results produced by a typical execution. The first two inputs specify a total of 16 citizens and a total of 16 distinct citizen-to-citizen relationships. Under "Citizens Friends" the 16 rows of output describe the network created by the zero-seeded random number generator in the Community constructor. The numbers in the first column are the IDs of the selected citizens. The sets of numbers in the second column are the IDs of friends of the citizens in the first column. In each case, friend IDs are in ascending order because each set of friends is an instance of the TreeSet class. Notice that all friendships are mutual. That is, if citizen B is a friend of citizen A, then citizen A is a friend of citizen B. This makes the total number of friends in the table (32) equal to twice the specified number of (bidirectional) relationships. Also, notice that different citizens have different numbers of friends, and citizen 7 has no friends at all.

The third input arbitrarily specifies a particular message originator: citizen 4. The two columns below that input describe message propagation. The first sender out of the sendersQueue queue (Citizen 4) sends the message to friends 1, 2, 5, and 10, who then enter the queue in that order. The next sender out of the queue (citizen 1) finds 4 is already informed, but she sends to 8, who then enters the queue. The next sender out of the queue (citizen 2) has no uninformed friends. The next sender out of the queue (citizen 5) finds that 2 and 4 already are informed, but he sends to 11. This process continues until the queue becomes empty. Because citizen 7 is nobody's friend (poor guy!), he is never informed.



```
/*********************************************************************
* Citizen.java
* Dean & Dean
*
* This represents an element in a network of citizens.
**********************************************************************/

import java.util.*;                                 // Set & TreeSet

public class Citizen
{
  private static int nextID = 0;                    // for unique IDs
  public final int ID = nextID++;                   // cannot change!
  private Set<Integer> friends = new TreeSet<>();
  private int delay;

  //*****************************************************************

  public void addFriend(int friendID)
  {
    this.friends.add(friendID);
  } // end addFriend

  public Set<Integer> getFriends()
  {
    return this.friends;
  } // end getFriends

  public void setDelay(int delay)
  {
    this.delay = delay;
  } // end setDelay

  public int getDelay()
  {
    return this.delay;
  } // end getDelay

  //*****************************************************************

  public String toString()
  {
    return String.format("%d\t%s", ID, friends);
  } // end toString
} // end class Citizen
```

Figure 10.18 Citizen class Used by CommunityDriver in Figure 10.16 and by Community in Figures 10.17a and 10.17b.



Sample session:

```
Enter citizen & relation quantities: 16 16
Citizen Friends
0       [8, 9, 15]
1       [4, 8]
2       [4, 5]
3       [9]
4       [1, 2, 5, 10]
5       [2, 4, 11]
6       [9, 10]
7       []
8       [0, 1]
9       [0, 3, 6, 12]
10      [4, 6]
11      [5, 13]
12      [9]
13      [11]
14      [15]
15      [0, 14]
Enter information source ID: 4
Citizen Delay
4       0
1       1
2       1
5       1
10      1
8       2
11      2
6       2
0       3
13      3
9       3
15      4
3       4
12      4
14      5
```

Figure 10.19 Typical output produced by the CommunityDriver program in Figure 10.16 In the graphical representation at right, solid lines are connections to uninformed friends. Dashed lines are connections to previously informed friends. Friendless Citizen 7 is never informed.

Page 475

Because the informedCitizen set returned by the spreadWord method is a LinkedHashSet, the sequence of sender IDs in the output is the same as the sequence of entries into the informedCitizen set. Notice that this sequence of outputs moves through the network row by row, going left to right across each row before moving down to the next row, until all citizens that can be reached have been reached. If you like fancy terminology, you'll be pleased to learn that this sequence is called a *breadth-first traverse*.

10.12 GUI Track: Second Iteration of Problem Solving with CRC Cards (Optional)

This section illustrates some Java Collections Framework interfaces and classes (Map and LinkedHashMap and

Set and LinkedHashSet) in a GUI context. The context is a second iteration of the CRC_Card program introduced in Chapter 8's optional GUI section. This second iteration employs Chapter 8's CRC_Card class in conjunction with a new and improved driver class, which enables the user to create and display multiple cards and later print any one of them.

Suppose the user executes the program with two arguments after the name of the class to be executed, like this:

```
> java CRCDriver2 CRC_Card CRCDriver2
```

The program immediately displays two labeled and named but otherwise blank CRC cards in a staggered stack in the upper left part of the computer screen. Each execution argument appears in two places on each card—as a stage title at the left end of the title bar and as a default classname in the text field after the CLASS label. Near the center of the screen the program also displays a ChoiceDialog (management dialog) with the title “CRC Card Management.”

The default choice for the management dialog is Quit. Other choices are Add, Show All, and Print Card. When Quit is visible in the management dialog and the user clicks that dialog’s OK button, the driver code deletes everything on the screen and terminates program execution. Clicking management dialog’s Cancel button or title bar X deletes everything on the screen and terminates program execution for any dialog choice.

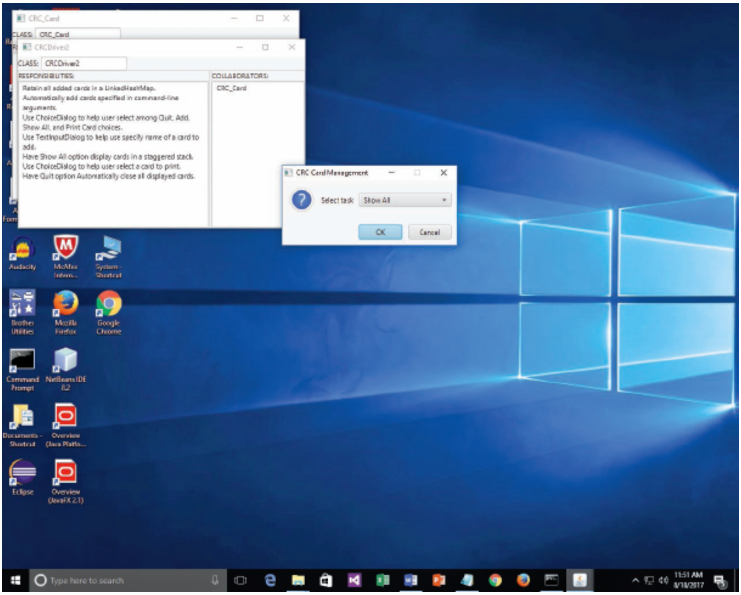
Whenever the management dialog is visible, the user can select its Add choice and then click OK. This makes a TextInputDialog (add dialog) appear with the request to enter a new card name. Assuming the user enters some text (not nothing or null) and then clicks this add dialog’s OK, the program creates a new CRC card with the user-entered text for both card name and default classname. Then the program automatically redisplays all cards (including the new one) in a staggered stack arrangement like that in Figure 10.20. This redisplay always stacks the cards in the order in which they were specified and/or added, starting with the first execution argument and ending with the last card added.

At any time, the user can alter the class name in the box after CLASS and can make text entries or alter previous text entries in the larger boxes under RESPONSIBILITIES or COLLABORATORS. Before doing this, the user may wish to hide the management dialog by temporarily clicking the little bar in its title bar, and then separate the CRC cards by dragging them to different places on the screen. Figure 10.20 shows what you might see after making entries under RESPONSIBILITIES and COLLABORATORS that are appropriate for a pair of CRC cards that describe the two classes in the present program, restoring to view a possibly hidden management dialog, and selecting its Show All option to repaint the screen with everything in its default location.

If the user clicks the title bar X on any individual CRC card, that card's image disappears. But the program remembers it along with all of its user-entered text information. Later the user can redisplay it along with all other CRC cards with their previous user-entered text information in the standard staggered stack arrangement by returning the management dialog to the screen, selecting its Show All choice and clicking OK.

Whenever the management dialog is visible, the user can select its Print

Card choice and click OK. This causes another ChoiceDialog (print dialog) to appear—with a blank default choice. Print dialog’s other choices will be all the CRC cards remembered by the program—including all their user-entered information. The selected card need not be currently visible on the computer screen. To obtain a paper printout of one of these cards with all previously user-entered text, the user selects that card’s name in the print dialog and then clicks print dialog’s OK.


CRC_Card
CRCDriver2
CRC Card Management
OK
Cancel
Audacity
McAfee Intern...
System - Shortcut
Brother Utilities
Mozilla Firefox
Google Chrome
Command Prompt
NetBeans IDE 8.2
Documents - Shortcut
Overview (Java Platfo...
Eclipse
Overview (JavaFX 2.1)
Type here to search
11:51 AM
8/18/2017

CLASS: CRC_Card

CLASS: CRCDriver2

RESPONSIBILITIES:	COLLABORATORS:
Retain all added cards in a LinkedHashMap. Automatically add cards specified in command-line arguments. Use ChoiceDialog to help user select among Quit, Add, Show All, and Print Card choices. Use TextInputDialog to help use specify name of a card to add. Have Show All option display cards in a staggered stack. Use ChoiceDialog to help user select a card to print. Have Quit option Automatically close all displayed cards.	CRC_Card

Select task: Show All

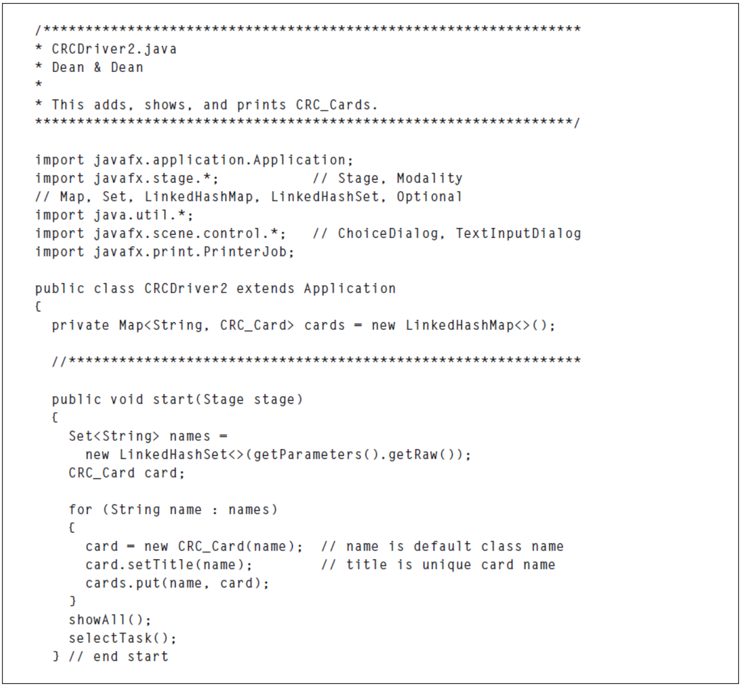
Figure 10.20 Typical display produced by second iteration of CRC_Card program after execution with two cards specified at execution and later RESPONSIBILITIES and COLLABORATORS entries by user Figure 8.19 has the driven class's code. Figures 10.21a through 10.21d have the new driver's code.

©JavaFX

Figure 10.21a shows the first part of this iteration's driver. JavaFX imports provide access to the Stage and Modality classes, the ChoiceDialog and TextInputDialog classes and the PrinterJob class. We'll use Modality to allow the user to jump back and forth between the management dialog and card stages. You may recall ChoiceDialog and TextInputDialog in the optional GUI section at the end of Chapter 3. The java.util package provides access to the Optional class used by JavaFX dialogs and to the Map, Set, LinkedHashMap, and LinkedHashSet classes. These last four classes are members of the Java Collections Framework described in previous sections of the current chapter.

CRCDriver2's only instance variable is a LinkedHashMap of CRC_Card objects accessed by key values that are the names of those cards. Each key must be unique, so the key values are a set. A LinkedHashMap is a relatively efficient Map that always iterates through its contents in the order of their insertion.

In the start method the first declaration loads a LinkedHashSet with the contents of the Parameters variable in our extension of the JavaFX Application class. Recall that every main method has a String[] args parameter. This args is an array of space-separated String words a user may append to the class name in the program execution command. As indicated in Chapter 3, any Java program's main method may contain code that uses these optional execution arguments. JavaFX has the same ability to accept optional space-separated String words appended to the class name in the program execution command. But instead of putting this data into a main method7 parameter, it puts it into a Properties variable. Our program copies command-line arguments into a List by having the extended Application object call getParameters().getRaw(). Then it puts this list's contents into a LinkedHashSet, which (like a LinkedHashMap) always iterates through its contents in the order of their insertion.

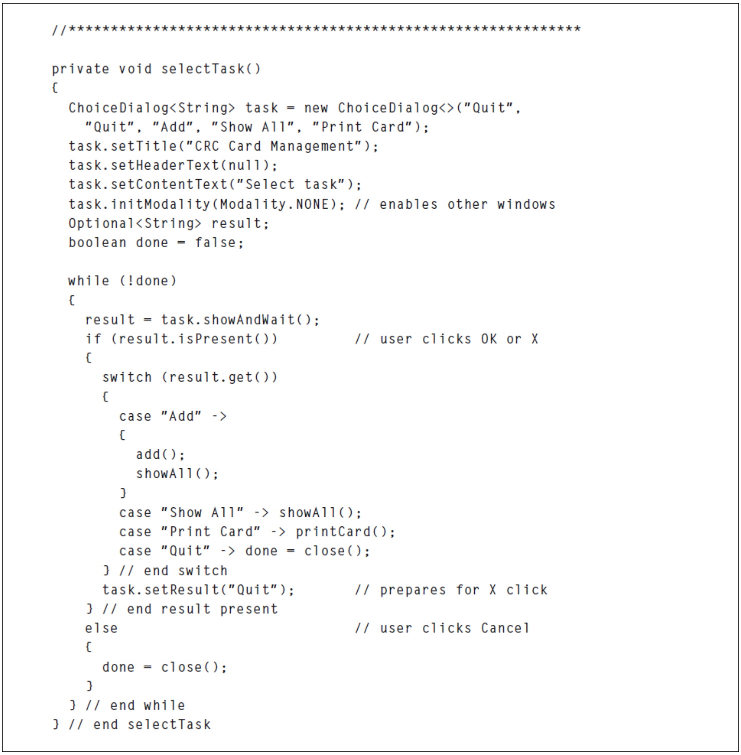


```
/*******************************************************************
* CRCDriver2.java
* Dean & Dean
*
* This adds, shows, and prints CRC_Cards.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.*;          // Stage, Modality
// Map, Set, LinkedHashMap, LinkedHashSet, Optional
import java.util.*;
import javafx.scene.control.*;  // ChoiceDialog, TextInputDialog
import javafx.print.PrinterJob;

public class CRCDriver2 extends Application
{
  private Map<String, CRC_Card> cards = new LinkedHashMap<>();

  //****************************************************************

  public void start(Stage stage)
  {
    Set<String> names =
      new LinkedHashSet<>(getParameters().getRaw());
    CRC_Card card;

    for (String name : names)
    {
      card = new CRC_Card(name);  // name is default class name
      card.setTitle(name);        // title is unique card name
      cards.put(name, card);
    }
    showAll();
    selectTask();
  } // end start
```

Figure 10.21a CRCDriver2 class that manages Figure 8.19's CRC cards—part A

Our start method's for-each loop creates a CRC_Card for each name in the LinkedHashSet and then inserts that card into the LinkedHashMap, with that card's name as the key. Then it shows all cards on the computer screen and calls the selectTask helper method in Figure 10.21b.

The selectTask method creates the management dialog stage at the center of Figure 10.20. The Modality.NONE specification allows the user to move back and forth between the management dialog and individual cards. The Optional result gets one of the four options delineated in the ChoiceDialog constructor. The while loop repeats until the boolean returned by the close method call in the else clause sets done to true. This happens whenever a result is present with the value "Quit".



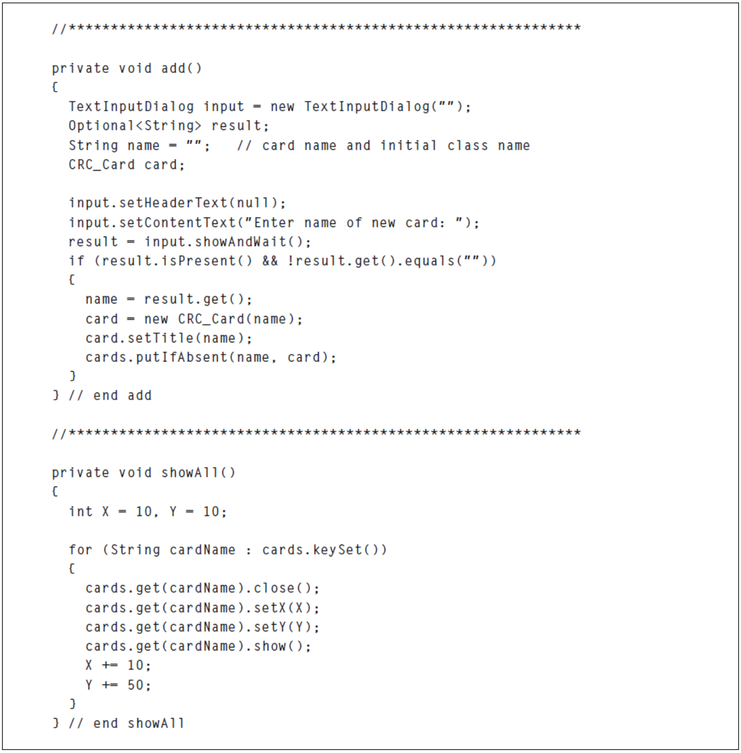
```
//*************************************************************

private void selectTask()
{
  ChoiceDialog<String> task = new ChoiceDialog<>(”Quit”,
    ”Quit”, ”Add”, ”Show All”, ”Print Card”);
  task.setTitle(”CRC Card Management”);
  task.setHeaderText(null);
  task.setContentText(”Select task”);
  task.initModality(Modality.NONE); // enables other windows
  Optional<String> result;
  boolean done = false;

  while (!done)
  {
    result = task.showAndWait();
    if (result.isPresent())         // user clicks OK or X
    {
      switch (result.get())
      {
        case ”Add” ->
        {
          add();
          showAll();
        }
        case ”Show All” -> showAll();
        case ”Print Card” -> printCard();
        case ”Quit” -> done = close();
      } // end switch
      task.setResult(”Quit”);       // prepares for X click
    } // end result present
    else                            // user clicks Cancel
    {
      done = close();
    }
  } // end while
} // end selectTask
```

Figure 10.21b CRCDriver2 class that manages Figure 8.19’s CRC cards—part B

In Figure 10.21b’s switch statement, note the method calls to add, showAll, printCard, and close subordinate helper methods. This switch statement does not need a default clause because its case clauses account for all ChoiceDialog options.

The only odd feature in this code is the task.setResult("Quit") statement after the switch statement. We include this apparently extra statement to handle the special situation where the user previously selected something other than "Quit" but now tries to quit by clicking the title bar X. When this occurs, a result is present but that result is whatever remains after the previous choice. The “extra” setResult method call always resets that residual result to the original "Quit" default.

%Figure 10.21c contains the subordinate helper methods add and showAll. The add method employs a TextInputDialog (add dialog) to ask the user for the name of a new card. If the user enters nothing, or clicks X or Cancel, the method does nothing and its dialog stage closes. If the user enters something and clicks OK, the method creates a card with a card name and default class name equal to the entered text. If the new card name differs from all previous card names,8 the putIfAbsent method call adds that card to the others in the LinkedHashMap. The showAll method iterates through the card names in LinkedHashMap's key and redisplays them in the program's standard staggered stack.

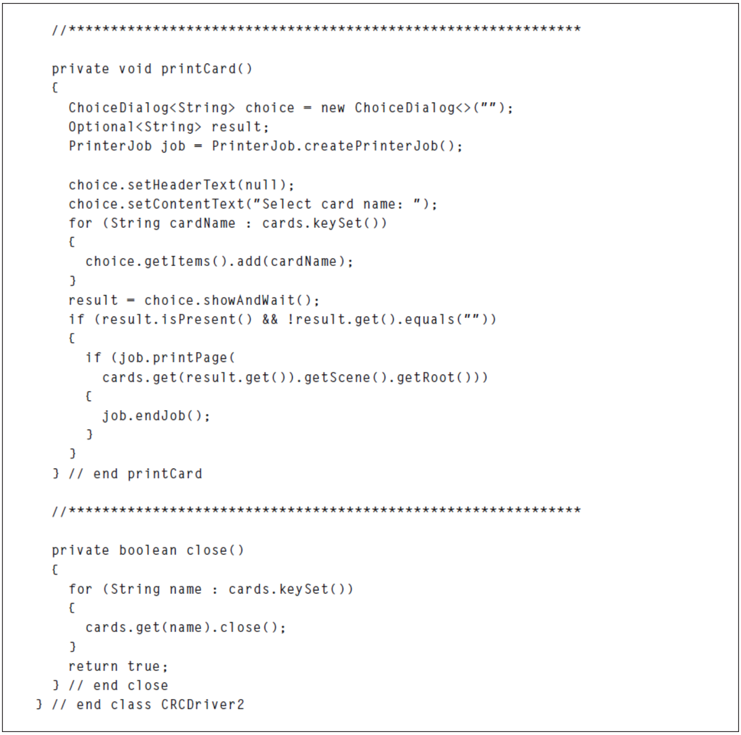


```
//*****************************************************************

private void add()
{
  TextInputDialog input = new TextInputDialog("");
  Optional<String> result;
  String name = "";    // card name and initial class name
  CRC_Card card;

  input.setHeaderText(null);
  input.setContentText(”Enter name of new card: ”);
  result = input.showAndWait();
  if (result.isPresent() && !result.get().equals(""))
  {
    name = result.get();
    card = new CRC_Card(name);
    card.setTitle(name);
    cards.putIfAbsent(name, card);
  }
} // end add

//*****************************************************************

private void showAll()
{
  int X = 10, Y = 10;

  for (String cardName : cards.keySet())
  {
    cards.get(cardName).close();
    cards.get(cardName).setX(X);
    cards.get(cardName).setY(Y);
    cards.get(cardName).show();
    X += 10;
    Y += 50;
  }
} // end showAll
```

Figure 10.21c CRCDriver2 class that manages Figure 8.19's CRC cards—part

C

Figure 10.21d contains the subordinate helper methods printCard and close. The printCard method employs another ChoiceDialog to ask the user to choose either the default nothing ("") or one of the cards currently in the LinkedHashMap. If the user selects something other than the default and clicks OK, The cards.get(result.get()) method call uses result's String value as a name key that retrieves a reference to the selected card from the LinkedHashMap. Then—in chained method calls—this reference retrieves a reference to that card's scene, and the latter reference returns a reference to the root object in that scene. This root object—a VBox called sceneGraph in the CRC_Card class—contains everything except the stage and scene boundary. This root object becomes the argument of the PrinterJob's printPage method call, and that's what the printer prints.



```
  //*************************************************************

  private void printCard()
  {
    ChoiceDialog<String> choice = new ChoiceDialog<>("");
    Optional<String> result;
    PrinterJob job = PrinterJob.createPrinterJob();

    choice.setHeaderText(null);
    choice.setContentText("Select card name: ");
    for (String cardName : cards.keySet())
    {
      choice.getItems().add(cardName);
    }
    result = choice.showAndWait();
    if (result.isPresent() && !result.get().equals(""))
    {
      if (job.printPage(
        cards.get(result.get()).getScene().getRoot()))
      {
        job.endJob();
      }
    }
  } // end printCard

  //*************************************************************

  private boolean close()
  {
    for (String name : cards.keySet())
    {
      cards.get(name).close();
    }
    return true;
  } // end close
} // end class CRCDriver2
```

Figure 10.21d CRCDriver2 class that manages Figure 8.19's CRC cards—part D

CRCDriver2's final close method is simple. It goes through all the keys in the LinkedHashMap, uses each key to retrieve a reference to its corresponding card object, and closes that card's screen image. Then it returns the boolean value true to what called it in the previously described selectTask method—the statement setting done in the switch statement's default clause or the statement setting done in the else clause that handles the result not being present. In either case, the close method's return statement terminates selectTask's while loop.

Summary

- If you need to insert or delete several elements within an array, instead of using an array, consider using an ArrayList. When you declare an ArrayList variable, you must include the element type in angled brackets after the ArrayList collection type name. If the declaring statement also instantiates, you may omit the element type in the subsequent instantiation, as in this ArrayList of Car elements:

ArrayList<Car> car = new ArrayList<>();

- If instantiation comes later, it should include the element type, like this:

car = new ArrayList<Car>();

- An ArrayList automatically adjusts its length to accommodate changes in the number of elements contained. Use ArrayList's add(E elem) to add an element to the end of an ArrayList, or use add(int index, E elem) to insert an element at a particular position. To remove an element from a particular position, use remove(int index). The size method returns the total number of currently contained elements.

- An ArrayList stores objects only.

Autoboxing and unboxing automatically make necessary conversions between primitives and wrapped primitives, so you don't have to worry about that. But if you want an ArrayList of primitives like int, you must declare it with the wrapped type, like this:

> ArrayList<Integer> num = new ArrayList<>();

- You can pass objects to and from methods anonymously.
- Because a LinkedList implements almost all of the methods implemented by an ArrayList, it can perform the same functions as an ArrayList. But instead of keeping its elements in an array, a linked list chains them together with forward and backward references.
- An interface serves as a template

for implementing classes that share common characteristics.

- You can use System.nanoTime method calls to measure and compare code performance.
- Chaining makes access operations slower in a LinkedList than in an ArrayList.
- A queue uses a first-in, first-out (FIFO) strategy for adding and removing elements.
- A stack uses a last-in, first-out (LIFO) strategy for adding and removing elements.
- You can use an ArrayDeque to implement both queues and stacks efficiently.
- Interfaces like List, Queue, Set, and Map organize methods in the Java collections framework into coherent groups.

Review Questions

§10.2 The ArrayList Class

1. How is an ArrayList more versatile than an array?

2. To avoid runtime errors, you must always specify the size of an ArrayList when you declare it. (T / F)

3. What is the return type of the ArrayList class's get method?

4. If you call the ArrayList method, add(i, x), what happens to the element originally at position i?

§10.3 Storing Primitives in an ArrayList

5. Specifically, under what circumstances does autoboxing take place?

6. Write one statement that

appends the double value, 56.85, to the end of an existing ArrayList called prices.

7. Write one statement that displays all of the values in an ArrayList of Doubles called prices. Put the complete list in square brackets and use a comma and a space to separate different values in the list.

§10.4 ArrayList Example Using Anonymous Objects and the For-Each Loop

8. What is an anonymous object?

§10.5 ArrayLists Versus Standard Arrays

9. Given:

- You have a WeatherDay class that stores weather information for a single day.
- You like to store WeatherDay

objects for a whole year.

- The primary task of your program is to create distinct sorted collections of WeatherDay objects (sorted by temperature, wind speed, and so on).

How should you store your WeatherDay objects—in an ArrayList or in a standard array? Provide a rationale for your answer.

§10.6 The LinkedList Class

10. Where does the JVM start the process of trying to access an element in the interior of a linked list?

§10.7 The List Interface

11. Based on what you have learned so far, what is a Java interface?

§10.8 Problem Solving: How to Compare Method Execution Times

12. For a given set of inputs, the execution time of a given program is always the same. (T / F)

13. When we remove an element from the center of an ArrayList, higher-index elements must shift downward. That shifting is not needed when we remove an element from a linked list. Why then does it usually take more time to remove an element from the center of a linked list?

§10.9 Queues, Stacks, and the ArrayDeque Class

14. Because an ArrayDeque is based on an array, when you remove an element from the front of an ArrayDeque, the computer must shift all higher elements down by 1. (T / F)

15. What is the difference between a queue and a stack?

§10.10 Overview of the Java Collections Framework

16. What are the interfaces at the tops of the two interface hierarchies in the Java collections framework?

17. What is a set?

§10.11 Collections Example—Information Flow in a Network of Friends

18. Why don’t we use a constructor parameter to establish the ID of a Citizen?

Exercises

---

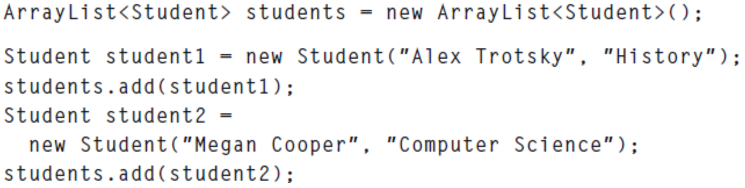
1. [after §10.2] What does the

ArrayList's set method do?

2. [after §10.3] A floating-point number can be thought of as a fraction. Provide a single statement (an initialization statement) that declares an ArrayList named fractions and assigns a newly instantiated empty ArrayList to it. The instantiated ArrayList should be able to store double values.

3. [after §10.3] Using the fractions ArrayList created in the previous exercise, provide a code fragment that assigns eight random numbers (generated by Math.random) into fractions. Use a standard for loop for this code fragment. Then provide an additional code fragment that uses a for-each loop to display the contents of the fractions ArrayList.

4. [after §10.4] Provide a more compact (but functionally equivalent) version of this code fragment:



```
ArrayList<Student> students = new ArrayList<Student>();

Student student1 = new Student(”Alex Trotsky”, ”History”);
students.add(student1);
Student student2 =
  new Student(”Megan Cooper”, ”Computer Science”);
students.add(student2);
```

5. [after §10.4] Suppose you have the ArrayList named students in Exercise 4, and assume the Student class has a getName method. Provide a for-each loop that prints all student names, one name per line.

6. [after §10.5] Modify the previous chapter’s Sort program using an ArrayList instead of an ordinary array. Instead of swapping, use ArrayList’s remove and add methods. Using nested for loops in the sort method makes the indexOfNextSmallest and swap methods unnecessary. In the following method skeleton, replace <*insert inner for loop here*> with appropriate code.

```
public static void sort(ArrayList<Integer> list)
{
  int indexNextSmallest;
  int nextSmallest;

  for (int i=0; i<list.size(); i++)
  {
    indexNextSmallest = i;
    nextSmallest = list.get(indexNextSmallest);
    // find smallest of unsorted values

    <insert inner for loop here>

    // move next smallest to end of unsorted values
    list.add(i, list.remove(indexNextSmallest));
  } // end loop through all
} // end sort
```

7. [after §10.6] Assume a Customer class has a constructor with a name parameter that initializes a name instance variable, and assume this class includes a getName method. A driver for this class includes this code:

```
LinkedList<Customer> customers = new LinkedList<>();
Customer friend = new Customer(”Pratima”);
int index;

customers.add(new Customer(”Pranoj”));
customers.add(new Customer(”Rachel”));
customers.add(friend);
customers.add(new Customer(”Mohammad”));
customers.add(new Customer(”Jasur”));
customers.add(new Customer(”Shyan”));
```

Add code that inserts into the

customers list a new customer named “Peter”. Make the insertion immediately before the friend object.

8. [after §10.6] Given the Customer class and driver described in the previous exercise, add code that removes from the customers list the discouraged customer who has been standing immediately behind (after) the friend customer and then generates the following output:

> <*discouraged customer’s name*> goes home

9. [after §10.8] The main method in the ListExecutionTimes program contains this local variable initialization statement:

> ArrayList<Double> list = new ArrayList<>();

After determining ArrayList execution times, we modified the program by replacing the above statement with this:

```
LinkedList<Double> list =
new LinkedList<>();
```

Declare the list variable in a way that allows you to assign an ArrayList object to it initially and then later assign a LinkedList object to it. Specifically, provide a declaration statement for list, an assignment statement for an ArrayList and an assignment statement for a LinkedList. Why can you assign either an ArrayList object or a LinkedList object to this same variable?

10. [after §10.8] The times measured by the ListExecutionTimes program includes not only the time required to perform the targeted get and set operations. It also includes time required to perform the for loop operations and the System.nanotime method calls. You can measure these overhead times by running the program with the get and set method calls removed. Modify this chapter's ListExecutionTimes class to measure the net average get and set time with the for loop overhead and nanotime method call overhead removed.

11. [after §10.9] Assume that a stack has been declared and instantiated with the ArrayDeque class. Show the output produced by the following code fragment:

```
stack.push('Z');
stack.push('Y');
System.out.println(stack.peek());
stack.push('X');
System.out.println(stack.pop());
stack.push('W');
while (!stack.isEmpty())
{
  System.out.println(stack.pop());
}
System.out.println(stack.peek());
```

You should be able to do this by just thinking about it, but feel free to complete and run a program to check your answer.

Page 486

12. [after §10.10] Assume an ordinary queue has been declared with the Queue interface and instantiated with the ArrayDeque class. Also, assume a priority queue has been declared with the Queue interface and instantiated with the PriorityQueue class. Figure 10.15 shows that the PriorityQueue class also implements the Queue interface. Therefore, the PriorityQueue class also implements add and remove methods. Its remove method does the same thing as ArrayDeque's remove method. But its add method is different. When it adds elements, a priority queue inserts them into the queue so that lower-valued elements are always closer to the front. In other words, in a priority queue, lower-valued elements have priority and "crash" the line. Given this understanding, show the output produced by the following code fragment:

```
ordinaryCustomers.add("Pranoj");
ordinaryCustomers.add("Rachel");
ordinaryCustomers.add("Pratima");
System.out.println(ordinaryCustomers.remove());
ordinaryCustomers.add("Mohammad");
ordinaryCustomers.add("Jasur");
ordinaryCustomers.add("Shyan");

while (!ordinaryCustomers.isEmpty())
{
  priorityCustomers.add(ordinaryCustomers.remove());
}

while (!priorityCustomers.isEmpty())
{
  System.out.println(priorityCustomers.remove());
}
```

You should be able to do this by just thinking about it, but feel free to complete and run a program to check your answer.

13. [after §10.10] A collection like ArrayList<Student> students could contain two or more references to the same object (same student enrolled in more than one course). Therefore, students.size() would not necessarily return the total number of distinct Student objects in a school's student body. However, all Java collection framework classes have constructors that create the current type of collection from any other type of collection. Given the declaration:

int totalStudents;

Provide a single statement that assigns to totalStudents the total number of distinct Student objects in the students collection.

14. [after §10.10] In the ListExecutionTimes program, the getIndices method fills an ArrayList with randomly ordered indices and then copies the ArrayList's values to an array. Write code for an alternative getIndices method that returns a similar result. It calls the zero-parameter add method to populate the ArrayList with sequentially ordered indices from the array, calls the shuffle method from the Collections class to shuffle the array, and then copies the ArrayList's values to an array.

15. [after §10.10] Enhance the HungerGames program. Create a Tribute class with three instance variables, name (a String), district (an int), and gender (a char). Provide this new Tribute class with a constructor to initialize its instance variables and with get methods to access them. Modify the HungerGames class by changing String deceased to Tribute deceased, making the tributes list a list of Tribute objects, and providing this additional local variable:

```
Map<String, Tribute> tributeMap = new HashMap<>();
```

Populate the tributes list with these objects:

```
Tribute("Jody", 3, 'f')

Tribute("Harve", 3, 'm',)

Tribute("Newt", 5, 'm')

Tribute("Claire", 6, 'f')

Tribute("Ruth", 10, 'f')
```

Then, in a for-each loop that iterates through the tributes list, use Map’s put method to put each tribute into tributeMap, using each tribute’s name as the key to the object that more completely describes that tribute. Instead of using Math.random to pick random indices, use the Collections class’s shuffle method to randomize the sequence of the tributes list. Then, in an ordinary for loop that iterates through all but the last element in the shuffled tributes list, do the following:

a) Use tributes.get(i) to get the next object in the tributes list.

b) Use Map's remove method to remove this object from the tributesMap map.

c) Display the three attributes of the removed object. After this for loop, in a print statement, call tributesMap.keySet() to display the keys of all remaining entries in tributesMap. Does the operation that removes an entry from the map also remove an element from the list?

16. [after §10.11] Modify the Community program by enabling the user to explore alternate community relationships. Specifically, modify the Community constructor so it asks the user to supply an integer seed number.

Review Question Solutions

1. With an ArrayList, you can insert and delete elements anywhere in the sequence, and the list length grows and shrinks dynamically.

2. False. Normally, you specify no size for an ArrayList when you declare it.

3. The get method's return type is E, which refers to the type of each element in the ArrayList.

4. The element that is originally at position i shifts to the next higher index position.

5. Autoboxing takes place when a primitive is being used in a place that expects a reference.

6. prices.add(56.85);

7. System.out.println(prices);

8. An anonymous object is an object that's instantiated but it's not stored in a variable.

9. You should store your WeatherDay objects in a standard array.

Rationale:

- There's no need for the array to grow or shrink since the size is fixed at 366 (and standard arrays have a fixed size).
- In sorting, you'll need to access the objects quite often (and access is easier with standard

arrays).

10. The process of accessing a linked list element starts from either of the two ends.

11. A Java interface is a template for designing classes that share certain behaviors.

12. False. Executions vary with hardware and within the same hardware from one run to another.

13. Compared to jumping directly to a center element in an ArrayList, it takes a relatively long time to step from one of the ends to a center element in a LinkedList.

14. False. After removing the element at the front of an ArrayDeque, instead of shifting all higher elements down by 1, the computer moves the location of the front up by 1.

15. A queue is a first-in-first-out (FIFO) structure. The end from which elements are removed is opposite the end to which elements are added. A stack is a last-in-first-out (LIFO) structure. The end from which elements are removed (popped) is the same as the end to which elements are added (pushed).

16. The interfaces at the tops of the two interface hierarchies in the Java collections framework are the Collection<E> interface and the Map<K,V> interface.

17. A set is a collection of distinct objects, with no duplications.

18. We don't use a constructor parameter to establish the ID of a Citizen because we want to make it impossible for two distinct citizens to have the same ID. We need distinct citizen IDs because we use citizen IDs as members of the keySet of the Java Map called citizens and as members of the Java Set called informedCitizens.

---

[1] *The Hunger Games* (New York: Scholastic Press, 2008), written by Suzanne Collins, describes a dystopian world in which each of 12 districts must provide two “tributes” (a girl and a boy, ages 12–18) to the central government. The tributes are placed in a wilderness arena where they are forced to compete in a game of survival.

[2] If you need a refresher on wrapper classes, see Chapter 5.

[3] Although stacking plates is fairly common, alternative practices exist. Teenagers with dinner chores have been known to place dishes randomly behind the couch, where they are cleaned later by Sophie the Samoyed.

[4] See "Collections Framework Overview," https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.

[5] These fancy arrows are Unified Modeling Language (UML) symbols. See Appendix 7.

[6] A map's key does not need to be just one independent variable. It could be two or more independent variables. And those variables could be different types of objects. To convert a combination of several different objects into a single key, have each of those objects call a universally available hashCode method, which returns an int. Then add those returned int's to form a single key for the combination. When that single int sum goes into one of Map's methods, Java automatically converts it into an Integer.

[7] A JavaFX program may include a

main method. However, if a main method is present, it must include a launch(args); statement to invoke the program's start method and pass possible command-line arguments to the Parameters variable. It's also possible to start JavaFX code from the main method of another class that does not extend Application. For example, if that other class is in the same directory or package as CRCDriver2, you could include
CRCDriver2.launch(CRCDriver2.class, args); in that other class's main method. However, since a program may call launch only once, we recommend against the cumbersome use of main with JavaFX.

[8] If you want a new card for a different version of an existing class, append a new version number to the original name when creating the new version's card, and in the new card's display change the text in the CLASS text field

back to the original class name. This enables the program to remember and display multiple versions of the same class.

Recursion

Objectives

- Understand the basic concept of recursion.
- Appreciate the importance of stopping conditions.
- Describe iterative calculations with recurrence relations.

- Use a trace to show exactly what happens in a recursion.
- Convert between iterative and recursive implementations.
- Learn how to evaluate short recursions by hand.
- Use recursion to binary-search an array of ordered data.
- Use recursion to merge-sort an array of unordered data.
- Use recursion to solve the Towers of Hanoi puzzle.
- Partition a GUI problem into Model, View, and Controller concerns.
- Use recursion to model biological trees as fractals.
- Estimate performance and become familiar with Big O notation.

Outline

11.1 Introduction 490

11.2 Guidelines for Writing a Recursive Method 491

11.3 A Recursive Factorial Method 492

11.4 Comparison of Recursive and Iterative Solutions 496

11.5 Recursive Method Evaluation Practice 500

11.6 Binary Search 503

11.7 Merge Sort 506

11.8 Towers of Hanoi 510

11.9 Problem Solving with Performance Analysis 514

11.10 GUI Track: Drawing Trees with a Fractal Algorithm (Optional) 517

11.1 Introduction

To understand this chapter, you need to know how to write your own multiple-method programs. As such, you need to have read up through the Interlude. For Sections 11.6 and 11.7, you need to know about arrays, so in addition to reading up through the Interlude, you also need to have read Chapter 9, Sections 9.1 through 9.8. Section 11.9 analyzes algorithms from Chapters 9 and 10 as well as from this chapter. For the GUI-track Section 11.10, you need to know about object-oriented programming, so you need to have read up through Chapter 6.

In prior chapters, we provided many examples of methods calling other methods, but we didn't show an example of a method calling itself. If you're not used to it, the idea of a method calling itself probably seems pretty bizarre. Although it’s not all that common for a method to call itself, the technique can be used for elegant solutions to problems that otherwise would be very difficult to solve. A method that calls itself is known as a *recursive method*, and a program is said to exhibit *recursion* if it relies on recursive method calls.

In a general sense, recursion is the concept of describing something such that the description relies on a smaller version of the original thing. The smaller version describes itself by relying on a still smaller version of itself, and so on and so on. Typically, recursion appears in pictures and in problem solutions. Figure 11.1 is a recursive picture. In the figure, do you see the newspaper’s picture? It’s a smaller version of the original picture. If you look closely, you should be able to recognize the newspaper’s picture as containing a still-smaller version of the original picture. At the end of this chapter, there's a section that describes how to draw a recursive picture using Java. But before that section, and even in that section, the chapter's primary focus is not about drawing recursive pictures, but about using recursion to solve problems. Let's begin by thinking about a recursive solution to a nonprogramming problem . . .

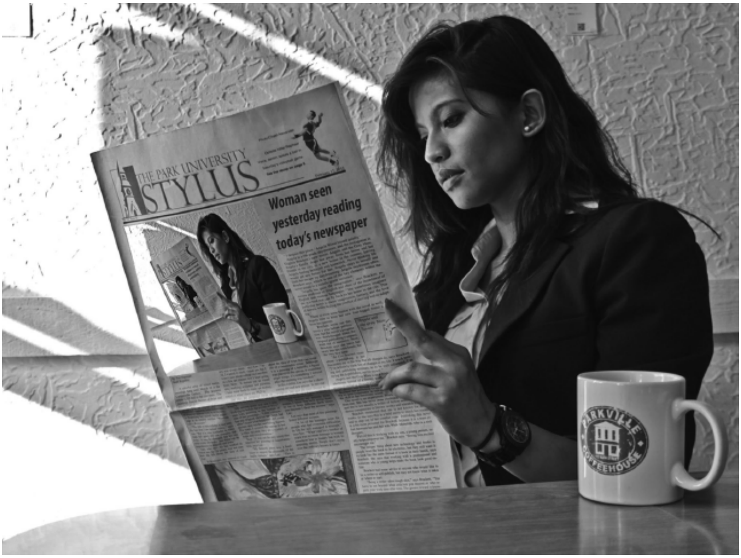

THE PARK UNIVERSITY
STYLUS
Woman seen yesterday reading today's newspaper
PARKVILLE COFFEEHOUSE

Figure 11.1 Recursive picture of a woman reading a newspaper

Courtesy of Rafael de Sant'Anna Neri.

Since the time of Greek antiquity, ships have relied on flags to relay information between them. In wartime, such flag signals have been particularly helpful. Unlike radio signals, which can be intercepted by the enemy from afar, flag signals can be read only by ships close enough to see them.

In battle, an admiral might want to determine the number of casualties his or her fleet has suffered. If the fleet's ships are in a line, with the admiral's ship at one end of that line, the admiral can initiate the process of determining the number of casualties by raising a flag (or set of flags) that signals a request for casualty information. That request propagates to the far end of the line. The last ship returns a flag signal that tells the previous ship the number of casualties that it has suffered. The previous ship then adds that number to its own number of casualties and uses another flag signal to relay this sum back to the previous ship. That continues until the total number of casualties gets back to the admiral's ship.

The naval flag-signaling process described here is recursive in that each ship has the same task— finding the total number of casualties suffered by its own ship plus the number of casualties suffered by the ships farther away. Another reason it's recursive is that each ship must rely on the next ship to gather a subset of the total information that's needed—the casualties suffered by the farther-away ships are a subset of the casualties suffered by all the ships. The flag-signaling process parallels the process of a method making a series of recursive method calls. Each method call has the same task, and each method call relies on the next method call to gather a subset of the final information that's needed.

Because beginning programmers often have a hard time wrapping their heads around recursion, in the first half of this chapter, we stick to the basics—the steps necessary to write a recursive method and a comparison of recursive methods and iterative/loop methods. In the interest of understandability, we use relatively short and simple programs to illustrate those basic concepts. In the second half of the chapter, we let you spread your wings by showing you recursive programs that better illustrate the true power and elegance of recursion. Those programs are more involved, but the effort is worth it. Without recursion, the programs would be very difficult to write. The programs implement algorithms that naturally lend themselves to recursion. Once you understand the recursive nature of the algorithms, the program implementations are fairly straightforward.

11.2 Guidelines for Writing a Recursive Method

In this section, we'll provide you with guidelines for writing recursive methods. Hopefully, they will make sense, but don't worry too much if they don't. They will make more sense in the next section, when you can use the guidelines as the basis for writing an actual recursive method.

If you're given a problem you think might lend itself to a recursive solution, the first step to finding such a solution is to figure out how the problem can be split into successively smaller subproblems. Each subproblem must solve the same type of problem as the original problem, but it must solve its problem using a smaller set of data than the data set for the previous problem. The next step is to identify a condition, called the *stopping condition* or *base case*, which indicates that the current problem is small enough such that it can be solved without having to subdivide the problem further. Can you think of the stopping condition in the naval flag-signaling procedure described earlier? It's when the last ship in the line receives the admiral's request. That last ship solves the problem of finding the number of casualties by returning just its own casualties. It doesn't need to worry about finding the number of casualties in ships farther away because there aren't any.

In implementing a recursive solution, you'll need to define a method with at least one parameter whose value becomes closer to the stopping condition's value with each successive recursive method call. If you don't have such a parameter, the recursive method calls will have no way of reaching the stopping condition. Within your recursive method's body, you'll need to use an if statement to check for the stopping condition. The if body should contain the solution to the simplest version of the problem. The else body should contain one or more calls to the same method with argument values that are closer to the stopping condition's value. When the method is called, the method continues to call itself recursively with argument values that become progressively closer to the stopping condition's value. The calls stop when the stopping condition is satisfied.

When the stopping condition is satisfied, the method solves the simplest version of the problem. Then the method returns the simplest problem's solution to the previous method execution at the next higher level of difficulty. That method execution generates the solution to its version of the problem. This process of returning simpler solutions to previous method executions continues back to the original method execution, which returns the solution to the original problem.

11.3 A Recursive Factorial Method

Calculating a Factorial—Background Details

In this section, we'll use the guidelines described in the previous section and come up with a recursive method that finds the factorial of a given number. Before working on the recursive method, let's review what a factorial is.

You can find the factorial of a number by multiplying all the numbers between 1 and the number itself. More formally, here is the formula for the factorial of a number n:

n! = n * (n-1) * (n-2) * ... * 2 * 1

The exclamation point is the mathematical symbol for factorial, so n! is “n factorial.” If n is 5, then

5! = 5 * 4 * 3 * 2 * 1 = 120

In coming up with a recursive method solution, the first step is to figure out how the factorial problem can be split into successively smaller factorial subproblems. In looking at the formula for the factorial of 5, notice that the right side of the equation is 5 times the factorial of 4. After all, isn’t 4 * 3 * 2 * 1 the factorial of 4? So 5 factorial can be written as

5! = 5 * 4!

Likewise, 4 factorial can be written as

4! = 4 * 3 * 2 * 1 = 4 * 3!

The pattern tells us that the factorial of any number n can be found by multiplying n times the factorial of n minus 1. Here is the formula:

n! = n * (n-1)!

This formula is the *recurrence relation* for a factorial. A recurrence relation splits up a problem into successively smaller subproblems. Finding the recurrence relation is the first step in solving a recursion problem.

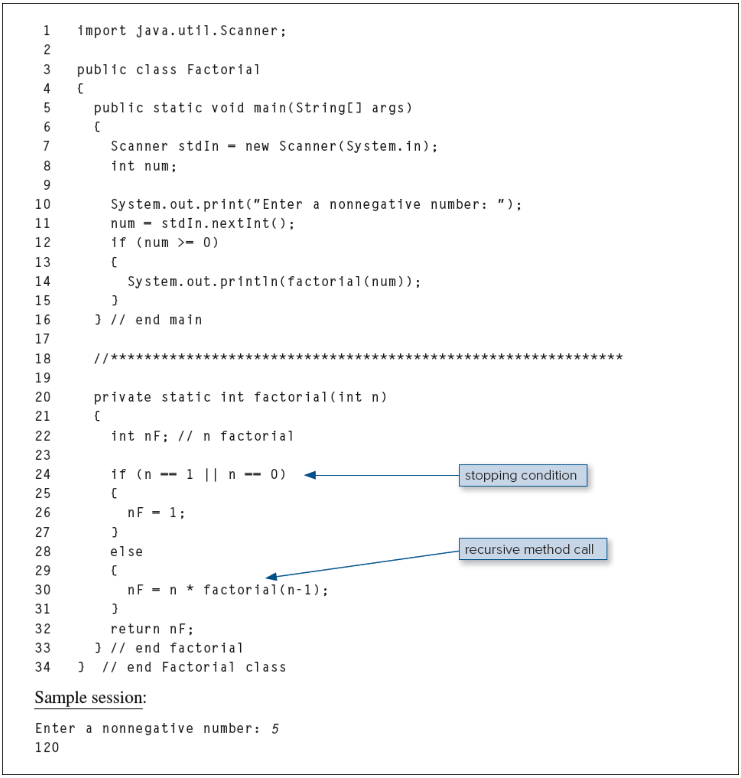
The next step is to identify the stopping condition. In calculating the factorial of a positive number n, the stopping condition is when n equals 1, because that's as low as you need to go when you're multiplying the numbers together. But you should also handle the case of calculating the factorial of 0. To do that, you should test for n equals 0 and return 1 in that case (because the factorial of 0 is 1). You don't need to find the factorial of a negative number because negative number factorials are undefined.

The Recursive Factorial Method

Look at Figure 11.2 and study its recursive factorial method. In the method's heading, note the parameter n. It's the number whose factorial is being found. Also, note the statement that contains the recursive method call:

```
nF = n * factorial(n-1);
```

factorial(n-1) is a recursive method call because it's inside a method named factorial, and it calls the factorial method. It calls the factorial method with an argument value of n-1 in order to match the recurrence relation described earlier: n! = n * (n-1)!. Note the n * in the Java code. Even though n * appears at the left of factorial(n-1), the JVM executes the recursive method call before performing the multiplication. Thus, the JVM performs the multiplication on the method call's returned value. It might seem strange to write a statement within a method X that relies on another execution of method X. That reliance requires you to make a leap of faith that the other execution of method X will work properly. Rest assured that it will work properly if the rest of the method is written properly. The leap of faith might be unsettling at first, but if you do it enough, hopefully you will learn to love the "magic of recursion." Remember the first time you rode a bike without training wheels and without someone guiding you? Scary, but after pedaling a while, you realized your momentum would keep you up. Recursion's leap of faith is kind of like that.



```
import java.util.Scanner;

public class Factorial
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int num;

    System.out.print(”Enter a nonnegative number: ”);
    num = stdIn.nextInt();
    if (num >= 0)
    {
      System.out.println(factorial(num));
    }
  } // end main

  //*************************************************************

  private static int factorial(int n)
  {
    int nF; // n factorial

    if (n == 1 || n == 0)
    {
      nF = 1;
    }
    else
    {
      nF = n * factorial(n-1);
    }
    return nF;
  } // end factorial
} // end Factorial class
```

Sample session:

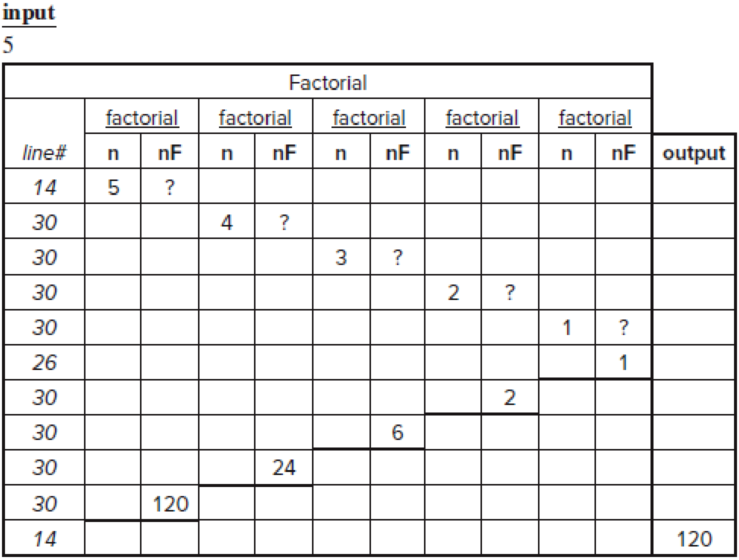
```
Enter a nonnegative number: 5
120
```

Figure 11.2 Use of recursion to calculate the factorial of an integer

As part of a properly written recursive method, you must have a stopping condition. Note the stopping condition in Figure 11.2. The if statement indicates that if n equals 1 or 0, the JVM assigns 1 to nF (where nF stores the factorial of n). That should make sense because 1! equals 1 and 0! also equals 1. The last statement of the factorial method returns nF, so the prior call to factorial receives the calculated factorial of n.

Tracing the Recursive Factorial Method

Figure 11.3 shows a trace of the Factorial program in Figure 11.2, with the user entering a 5 in response to the “Enter a nonnegative number:” prompt. Looking at the program, you can see that on line 14, the main method calls the factorial method with an argument of 5. The next four calls to the factorial method are from within the factorial method on line 30 with arguments of 4, 3, 2, and 1, respectively. When the argument 1 is passed to factorial, the stopping condition is satisfied, and nF gets assigned the value 1 on line 26.



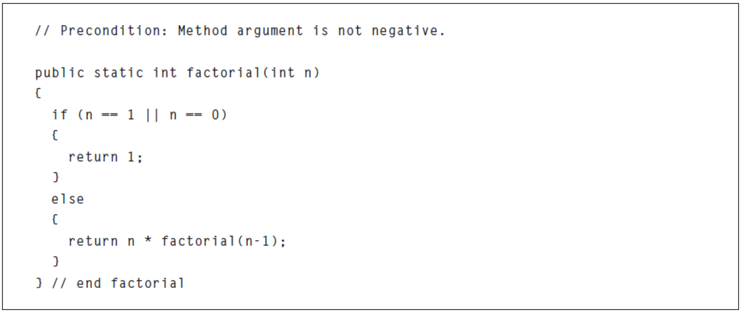
input
5

	Factorial										
	factorial		factorial		factorial		factorial		factorial		
line#	n	nF	n	nF	n	nF	n	nF	n	nF	output
14	5	?									
30			4	?							
30					3	?					
30							2	?			
30									1	?	
26										1	
30								2			
30						6					
30				24							
30		120									
14											120

Figure 11.3 Trace of the Factorial program in Figure 11.2

Then the returning process commences. The 1 from the fifth factorial call returns to the fourth - factorial call, which, on line 30, multiplies it by 2 and returns 2. This value returns to the third factorial call, which, on line 30, multiplies it by 3 and returns 6. This value returns to the second factorial call, which, on line 30, multiplies it by 4 and returns 24. This value returns to the first factorial call, which, on line 30, multiplies it by 5 and returns 120. This value returns to the argument of the println statement in the main method on line 14, and this statement prints out the computed value. In this problem, all the useful work is done in the return sequence, after the stopping condition is reached.

We included the local variable nF in the program in Figure 11.2 just to give this trace some substance. Hopefully, it helps you visualize the recursive calling that drills down to the simplest case and the subsequent result accumulation as the nested methods return. In practice, however, experienced programmers would not include the local nF variable. Instead, they would probably write the factorial method as in Figure 11.4.



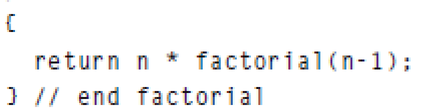
```
// Precondition: Method argument is not negative.

public static int factorial(int n)
{
  if (n == 1 || n == 0)
  {
    return 1;
  }
  else
  {
    return n * factorial(n-1);
  }
} // end factorial
```

Figure 11.4 Cleaner implementation of the factorial method

Infinite Recursive Loops

In writing a recursive method, beginning programmers sometimes focus exclusively on the method’s recursive method call and forget to include a stopping condition. Depending on the nature of the method, that can lead to an infinite recursive loop. An *infinite recursive loop* is when a method calls itself repeatedly without stopping. Each call requires the JVM to save information (such as the state of the program right before the call executes) in a special place in the computer’s memory—the *call stack*.

To prevent the call stack from eating up a computer's entire memory, the call stack's size is limited. If you run a program with an infinite recursive loop, the call stack eventually will fill up and generate a *stack overflow* runtime error and crash your program. If you'd like to experience the excitement of such a crash, substitute the following code for the factorial method in Figure 11.2's program and run the program:



```
public static int factorial(int n)
{
  return n * factorial(n-1);
} // end factorial
```

11.4 Comparison of Recursive and Iterative Solutions

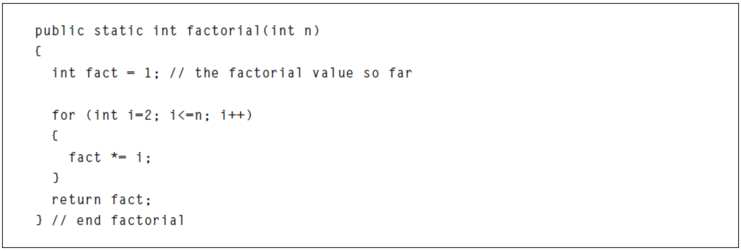
An Iterative Factorial Method

In the prior section, we first presented the factorial of n as

$$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$$

In looking at the repetitive nature of the above formula (it's a series of multiplications), you might have figured out that you can use a regular loop to calculate the factorial, and there's actually no need to use recursion. We used recursion not because we had to, but because we wanted a simple problem with a simple recursive solution. Recursion can be difficult to grasp initially, and it's best to start learning it with a relatively easy problem, like finding the factorial. Learn to walk before you run.

Anyway, now that you've "learned to walk" with recursion, let's take a step back and show an iterative/loop implementation for a factorial method. Figure 11.5 shows such a method. Instead of having a recursive method call at the bottom of the method, as shown in Figure 11.4, Figure 11.5's method has a loop that multiplies the numbers 1 through n, with one multiplication operation per loop iteration. We could have mimicked the recursive implementation by starting the multiplication at n and going down to 1, but we felt that starting at 1 and going up to n made more sense. In Figure 11.5, can you figure out what happens if the parameter n is either 0 or 1? The for loop never executes, and the method returns 1. That should make sense when you realize that 0! is 1 and 1! is also 1.



```
public static int factorial(int n)
{
  int fact = 1; // the factorial value so far

  for (int i=2; i<=n; i++)
  {
    fact *= i;
  }
  return fact;
} // end factorial
```

Figure 11.5 Iterative implementation of the factorial method

Characteristics of Recursive Solutions

Recursion does not add unique functionality. As exemplified by the factorial example shown above, all recursive programs can be converted to iterative programs that use loops instead of recursive method calls. So why use recursion? Because with some problems, a recursive solution is easier to understand. For example, some mathematical concepts, like the factorial of a number, are defined recursively, and they lend themselves well to programmatic solutions that use recursion. Many puzzles are easiest to solve with recursive thinking, and they also lend themselves well to programmatic solutions that use recursion. Examples are the Towers of Hanoi problem, which we will describe later in this chapter, and maze traversals.

Be aware that there is a downside to recursion. Recursive programs tend to be slow because they generate lots of method calls and method calls have lots of *overhead*. Overhead is work that the computer must do beyond the work described by program code. For each method call, the computer has to (1) save the calling module's local variables, (2) find the method, (3) make copies of call-by-value arguments, (4) pass the arguments, (5) execute the method, (6) find the calling module, and (7) restore the calling module's local variables. Whew! All that work takes time. That's why some recursive implementations can be prohibitively slow. When there a very large number of recursive calls, you should consider rewriting the solution with a loop implementation.

There are different types of recursion. *Mutual recursion* is when two or more methods call each other in a cycle of method calls. For example, if method A calls method B, method B calls method C, and method C calls method A, that's mutual recursion. You should be aware of mutual recursion, but we won't bother to show a mutual recursion program because such programs are not all that common. The most common type of recursion is when a method calls itself. If a method's body includes two (or more) recursive calls to itself and the method executes both calls, then the method is said to exhibit *binary recursion*. You'll see examples of binary recursion programs later in this chapter.

If a method executes just one recursive call to itself, then the method is said to exhibit *linear recursion*. Because it executes just one recursive call to itself, Figure 11.4's factorial method exhibits linear recursion. *Tail recursion* is a special case of linear recursion. It's when a recursive method executes its recursive call as its very last operation. You'll see an example of tail recursion in the upcoming subsection, but let's first examine the factorial method more closely. It does not exhibit tail recursion. Because its recursive call is positioned at the bottom of its method body, the factorial method might appear to exhibit tail recursion. However, it's not just the recursive call's position that matters; it's the recursive call's order of operation that matters. Here is the factorial method's recursive call statement:

```
return n * factorial(n-1);
```

In executing the statement, the JVM first calls factorial recursively and then multiplies n times factorial's returned value. So the last operation is multiplication, not the recursive call.

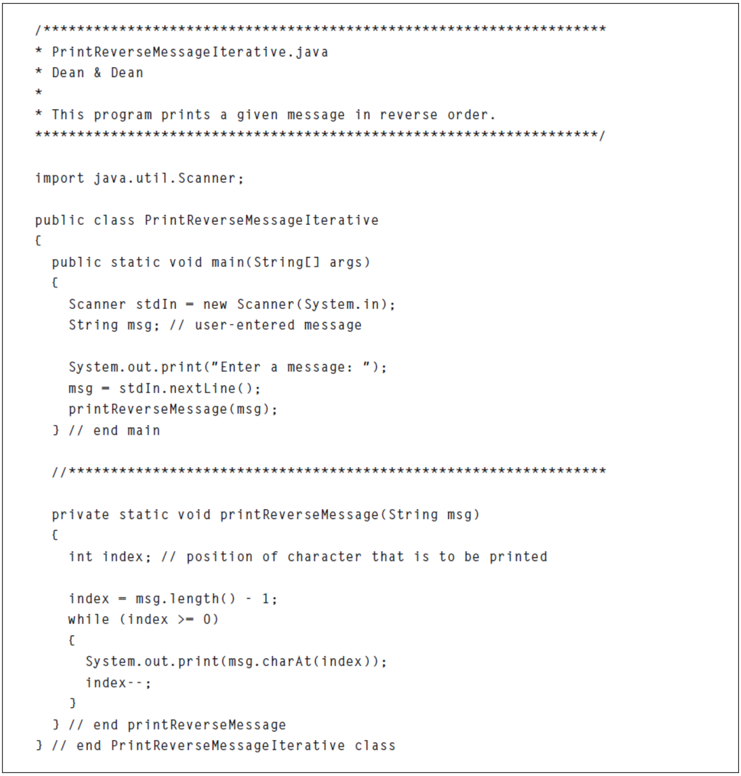
Of all the different kinds of recursions, tail recursions are the easiest to convert to loop implementations: Just use the changing recursive parameter as the loop index, and use the recursive stopping condition as the loop termination condition. That technique should become more evident as you study the tail recursion example in the next subsection. . . .

Converting from an Iterative Method to a Recursive Method

In the factorial discussion above, you saw an example of converting from a recursive method to an iterative method. Now let's go the other way—converting from an iterative method to a recursive method.

Note the printReverseMessage method in Figure 11.6. It receives a string parameter msg and prints the string in reverse order. So "Hedgehogs rock!" would print as "!kcor sgohegdeH." It uses a loop to traverse through each of the characters in msg, starting with the rightmost character and ending with the leftmost character. Now the challenge is to rewrite that iterative-implementation method as a recursive method.

As mentioned earlier, the first step to writing a recursive method is to figure out how the problem can be split into successively smaller subproblems. After printing a message's rightmost character, what subproblem remains to be solved? Printing the other characters in the message, of course! Solving that subproblem requires working on a substring of the original message—the substring that extends from the message's leftmost character to the message's penultimate character.[1] Before moving on to the next paragraph, try to use that explanation as the basis for writing a recursive printReverseMessage method on your own.



```
/*******************************************************************
* PrintReverseMessageIterative.java
* Dean & Dean
*
* This program prints a given message in reverse order.
*******************************************************************/

import java.util.Scanner;

public class PrintReverseMessageIterative
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String msg; // user-entered message

    System.out.print(”Enter a message: ”);
    msg = stdIn.nextLine();
    printReverseMessage(msg);
  } // end main

  //*****************************************************************

  private static void printReverseMessage(String msg)
  {
    int index; // position of character that is to be printed

    index = msg.length() - 1;
    while (index >= 0)
    {
      System.out.print(msg.charAt(index));
      index--;
    }
  } // end printReverseMessage
} // end PrintReverseMessageIterative class
```

Figure 11.6 Iterative implementation of printing a string in reverse order

The following code fragment is a reasonable first attempt at writing a recursive printReverseMessage method:

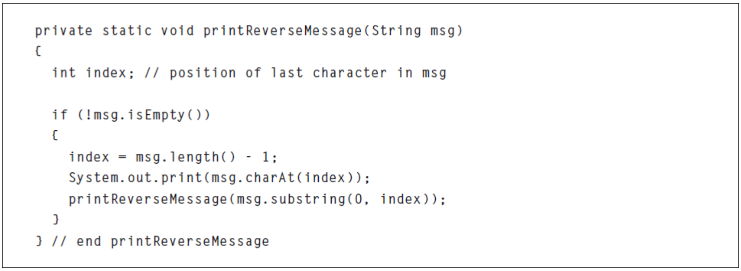
```
private static void printReverseMessage(String msg)
{
  int index = msg.length() - 1;
  System.out.print(msg.charAt(index));
  printReverseMessage(msg.substring(0, index));
}
```

Note how index is assigned the position of the message's last character. That index value is then used to print the message's last character. Note the substring method call. Remember how the substring method call works? It returns a substring of its calling object string (msg in this case), such that the returned string spans from the position of its first argument (0 in this case) to the position just to the left of its second argument (index in this case). Thus, the third line calls printReverseMessage with a substring argument that extends from the message's leftmost character to the message's penultimate character. That is our goal for the recursive call.

Are we done? Is this recursive printReverseMessage method complete? To answer that question, think about what happens when you call it with a msg value of "cow." The

JVM prints the w and calls the method recursively with an argument value of “co.” The JVM then prints the o and calls the method recursively with an argument value of “c.” The JVM then prints the c and calls the method recursively with an argument value of "", the empty string. The JVM then assigns −1 to the index variable and crashes when it attempts to execute msg.charAt(-1).

So what’s the problem, and what’s the fix? The method needs a stopping condition that stops the recursion when the msg parameter has shrunk all the way down to the empty string. See the corrected recursive printReverseMessage method in Figure 11.7. In particular, note the method’s if statement. If msg is non-empty, the JVM prints msg’s last character and calls printReverseMessage recursively. On the other hand, if msg contains the empty string, then the JVM does not call printReverseMessage recursively. It just returns. More often than not, recursive methods use if else statements, but in this case, there’s no need to do anything special when msg contains the empty string, so no else block is necessary.



```
private static void printReverseMessage(String msg)
{
  int index; // position of last character in msg

  if (!msg.isEmpty())
  {
    index = msg.length() - 1;
    System.out.print(msg.charAt(index));
    printReverseMessage(msg.substring(0, index));
  }
} // end printReverseMessage
```

Figure 11.7 Recursive implementation of the printReverseMessage method

Because printReverseMessage is a recursive method, after the recursion process hits its stopping condition, it returns (same as for all recursive processes). But what does it return? Since it’s a void method, it returns nothing! This is like a navy admiral issuing a command for all ships to turn right by 90 degrees. “No discussion. Just do it!” And no information comes back.

The recursive printReverseMessage method executes its recursive call as its very last operation, and, as such, it exhibits tail recursion. With tail recursion, the JVM performs its useful work while proceeding forward with the recursive calls. For the printReverseMessage method, its "useful work" consists of printing a character. On the other hand, the recursive factorial method does not execute its recursive call as its last operation, and the JVM performs its useful work (multiplication) while returning from the recursive calls.

In more complicated problems, you'll see that sometimes some operations occur before recursive calls (and take place during the calling process) and other operations occur after recursive calls (and take place during the return process). In problems like these, the recursive algorithms are usually much easier to understand and implement than their iterative counterparts.

11.5 Recursive Method Evaluation Practice

Page 5

Perhaps the best way to improve your feeling for recursion is to work through some problems by hand. You did this sort of thing before when you evaluated expressions in Chapter 3, Section 3.16. There, you wrote an expression and then repeated that expression on subsequent lines, with numerical values gradually substituted for variables or functions on lines above them.

Recursive Algorithms Expressed as Functions

When evaluating a recursive method

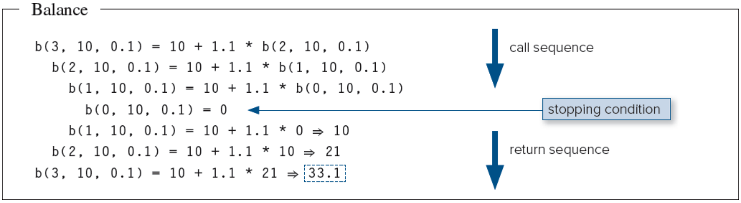
by hand, we'll often rewrite the method as if it were a mathematical function and use a one-letter abbreviation for the method name. For example, consider the Factorial program in Figure 11.2. In tracing the method call factorial(5), we'll rewrite it as f(5), where f stands for the factorial function. Using mathematical function notation, here's a compact description of the factorial algorithm:

$$b(n, D, R) = \begin{cases} D + (1 + R) * b(n-1) & \text{for } n >= 1 \\ 0 & \text{for } n < 1 \end{cases}$$

Notice that this algorithmic specification provides both the recurrence relation and the stopping condition together.

To evaluate a recursive method call by hand, start by writing the method with function notation (as shown above). For the first line of your recursion trace, write the recurrence relation with the variables replaced by the initial numbers. Under that, write the recurrence relation for the first subordinate method call with the variables replaced by appropriately altered numbers on both the left and the right sides of the equations. Continue like this until you reach a stopping condition. During the calling sequence, values that might be returned by recursive method calls are not yet known. They correspond to the question marks in Figure 11.3. The stopping condition produces a completely known value. On subsequent rows, rewrite what you previously wrote on the rows above, but in reverse order, replacing right-side unknowns with known values as you go. Eventually, this produces a completely known value for the original recursive call.

Here’s how the procedure above would be used to evaluate the factorial method being called with n equal to 5:



Balance

```
b(3, 10, 0.1) = 10 + 1.1 * b(2, 10, 0.1)
  b(2, 10, 0.1) = 10 + 1.1 * b(1, 10, 0.1)
    b(1, 10, 0.1) = 10 + 1.1 * b(0, 10, 0.1)
      b(0, 10, 0.1) = 0
    b(1, 10, 0.1) = 10 + 1.1 * 0 ⇒ 10
  b(2, 10, 0.1) = 10 + 1.1 * 10 ⇒ 21
b(3, 10, 0.1) = 10 + 1.1 * 21 ⇒ 33.1
```

The dashed box at the bottom identifies the desired answer.

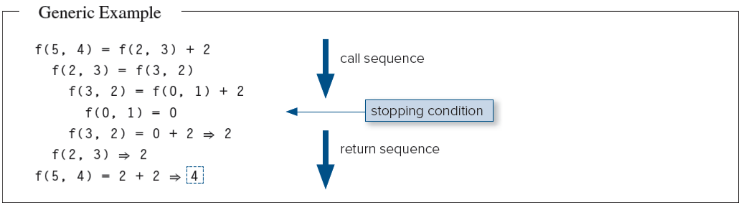
Page 501

For another example, consider a method that returns the balance, b, in a bank account after n equal periodic deposits of amount D, with interest rate R for the time period between deposits. Borrowing from the Java convention, we use lowercase letters for variables and uppercase letters for constants. Here's a compact description of the algorithm, where b stands for the balance function:

$$f(x, y) = \begin{cases} f(x-3, y-1) + 2 & x > 0, x > y \\ f(y, x) & x > 0, x <= y \\ 0 & x <= 0 \end{cases}$$

Let's say each deposit is in the amount D = 10, and the interest rate is R = 0.1. Suppose that we want to know the balance when n = 3. The evaluation looks like this:

Generic Example



```
f(5, 4) = f(2, 3) + 2
  f(2, 3) = f(3, 2)
    f(3, 2) = f(0, 1) + 2
      f(0, 1) = 0
    f(3, 2) = 0 + 2 ⇒ 2
  f(2, 3) ⇒ 2
f(5, 4) = 2 + 2 ⇒ 4
```

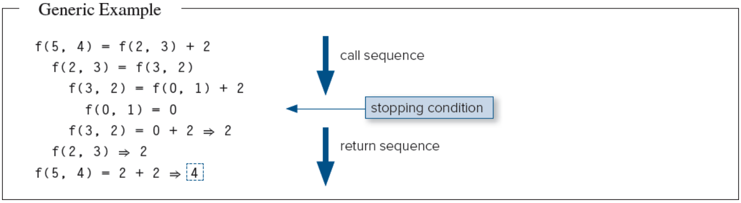
More Practice

To improve your recursive method call evaluation skills, you need to practice. In providing you with additional practice problems, we'll cut to the chase and start each problem with a mathematical-function notation. (Every non-void Java method is a mathematical function.) Be aware that in the real world, in evaluating a recursive method call, you'll have to come up with the mathematical-function notation on your own, complete with a recurrence relation and a stopping condition.

Here's a practice problem whose function has two parameters—x and y:

$$f(x, y) = \begin{cases} f(x-3, y-1) + 2 & x > 0,\ x > y \\ f(y, x) & x > 0,\ x <= y \\ 0 & x <= 0 \end{cases}$$

The function is named *f*, where *f* is a generic name for a function. To see how it works, let’s evaluate *f*(5, 4):

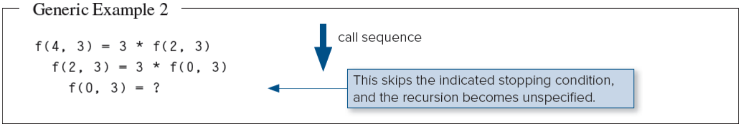


```
f(5, 4) = f(2, 3) + 2
  f(2, 3) = f(3, 2)
    f(3, 2) = f(0, 1) + 2
      f(0, 1) = 0
    f(3, 2) = 0 + 2 ⇒ 2
  f(2, 3) ⇒ 2
f(5, 4) = 2 + 2 ⇒ 4
```

In this example, the evaluation encountered an adequate stopping condition. But sometimes stopping conditions are not adequate. For example, suppose that you have this recurrence relation:

$$f(x, A) = \begin{cases} A * f(x-2, A) & x > 1 \\ A & x = 1 \end{cases}$$

Notice how this notation does not distinguish between assignment and equality. The equals sign to the left of the brace represents an assignment. The equals sign on the right represents equality. This kind of recursive specification expects you to make that distinction yourself. And that’s OK. But there is still something wrong. What is it? Here's how the evaluation of *f*(4, 3) would go:



Generic Example 2

```
f(4, 3) = 3 * f(2, 3)
  f(2, 3) = 3 * f(0, 3)
    f(0, 3) = ?
```

This skips the indicated stopping condition, and the recursion becomes unspecified.

Because x decrements by two in each iteration, when the starting value for x is even, it misses the indicated stopping condition. This problem is insidious because the indicated stopping condition works for odd starting values. Also, what if the starting x is a floating-point number? In general, inequality provides a better stopping condition than equality.

Here's another example:

$$f(x) = \begin{cases} f(x/2) & x > 0 \\ 0 & x <= 0 \end{cases}$$

The inequality takes care of cases that might skip zero. So is this OK? Suppose you start with x = 4. How many iterations would it take to reach the stopping condition? Let's see, 4, 2, 1, 0.5, 0.25, 0.125, . . . Hey, it never gets there! When you have this kind of problem, you might change that stopping condition to something slightly larger than zero, perhaps something like x < = 0.000001.

Here’s another example:

$$f(x) = \begin{cases} f(x) + 1 & x < 3 \\ 4 & x >= 3 \end{cases}$$

Up until this last example our adequate stopping conditions have been some kind of minimum. The stopping condition does not need to be a minimum. It could also be a maximum. So there’s nothing wrong with the fact that this stopping condition is a maximum. But there is still something wrong. What is it? The function gets larger, but the stopping condition does not look at the function. It just looks at x, and x never changes.

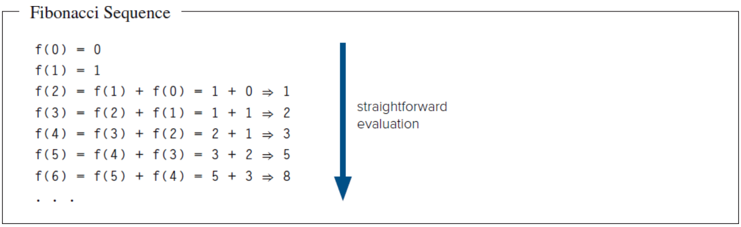
Implementing a Recurrence Relation with a Loop

Some recurrence relations lend

themselves to loop implementations where you start the loop from the stopping condition and move forward from there. As shown earlier in Figure 11.5, calculating a factorial lends itself well to a loop solution. Another example is a Fibonacci sequence, which a mathematician might describe with a recurrence relation like this:

$$f(n) = \begin{cases} f(n-1) + f(n-2) & n > 1 \\ 1 & n = 1 \\ 0 & n = 0 \end{cases}$$

As with the preceding examples, you could use this to implement a recursive solution and then convert that solution to a loop solution as we did in Section 11.4. But most humans would skip the recursive step. They would use the recursive relation's stopping conditions as initial values and apply its general formula like this:



Fibonacci Sequence

```
f(0) = 0
f(1) = 1
f(2) = f(1) + f(0) = 1 + 0 ⇒ 1
f(3) = f(2) + f(1) = 1 + 1 ⇒ 2
f(4) = f(3) + f(2) = 2 + 1 ⇒ 3
f(5) = f(4) + f(3) = 3 + 2 ⇒ 5
f(6) = f(5) + f(4) = 5 + 3 ⇒ 8
. . .
```

Using the calculations above as a pattern, here’s the loop implementation for finding the Fibonacci value at position n. We’re assuming that fibonacci is an array that has at least n + 1 elements.

```
fibonacci[0] = 0;
fibonacci[1] = 1;
for (int i=2; i<=n; i++)
{
  fibonacci[i] = fibonacci[i-1] + fibonacci[i-2];
}
```

11.6 Binary Search

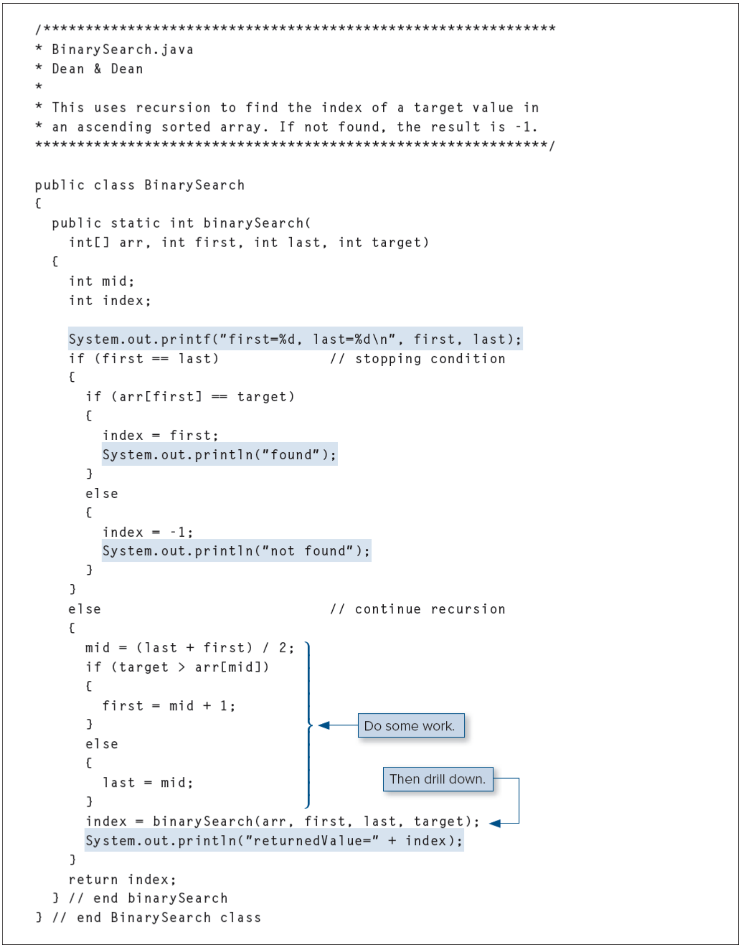
Now that you have learned how to work with several forms of recursion, you’re ready for some longer recursive programs. We hope that you’ll be able to appreciate that their solutions are more elegant than if the programs were implemented with loops. In this section's example, a single recursive call occurs after all the useful work is done. So it's another example of tail recursion.

Here's the problem: Suppose that you want to find the location of a particular value in an array. This is a common database operation. If the array is not sorted, the best you can do is use a sequential search and look at each item individually. If the array is very short, a sequential search is also the fastest way to search because a sequential search is very simple. If the array is long, however, and if it's relatively stable, it's often faster to sort the array and then use a *binary* search.

Back in Section 9.7, in Figure 9.12, we showed how to implement a binary search with a while loop. Now we'll show you how to do it with recursion. This will enable you to compare a recursive implementation with an iterative implementation. Figure 11.8

shows our implementation of this algorithm. We have included shaded print statements at appropriate places in the code to show what the code is doing when it executes. After debugging the program, you would want to remove all of these shaded print statements.

Figure 11.9 shows a driver that demonstrates the binary search algorithm implemented in Figure 11.8.

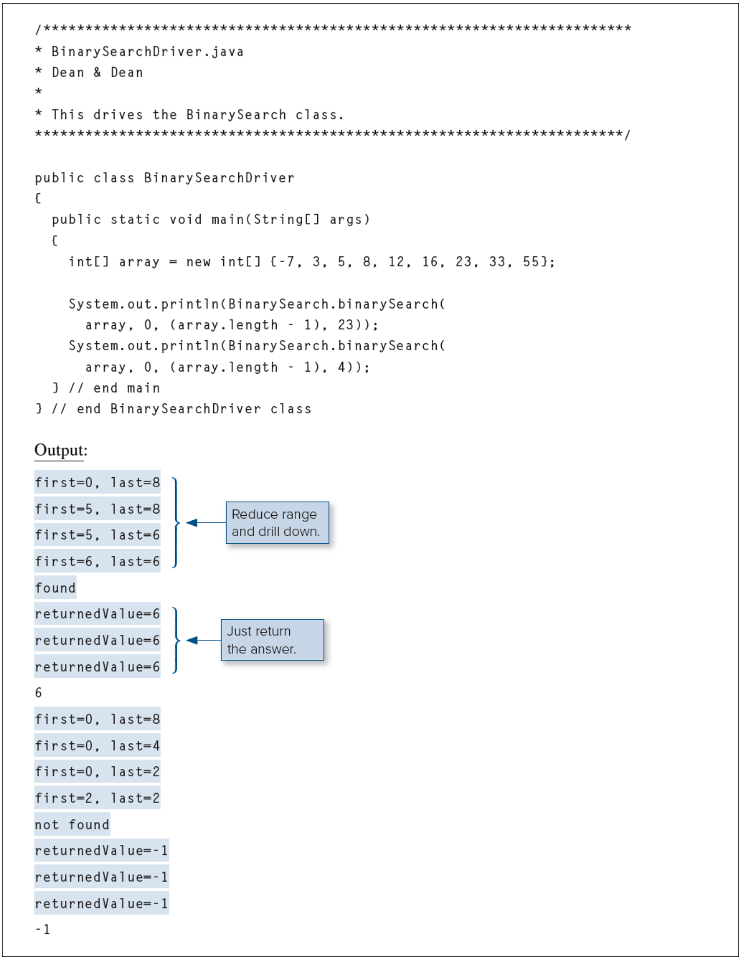


```
/***************************************************************
* BinarySearch.java
* Dean & Dean
*
* This uses recursion to find the index of a target value in
* an ascending sorted array. If not found, the result is -1.
***************************************************************/

public class BinarySearch
{
  public static int binarySearch(
    int[] arr, int first, int last, int target)
  {
    int mid;
    int index;

    System.out.printf("first=%d, last=%d\n", first, last);
    if (first == last)             // stopping condition
    {
      if (arr[first] == target)
      {
        index = first;
        System.out.println("found");
      }
      else
      {
        index = -1;
        System.out.println("not found");
      }
    }
    else                           // continue recursion
    {
      mid = (last + first) / 2;
      if (target > arr[mid])
      {
        first = mid + 1;
      }
      else
      {
        last = mid;
      }
      index = binarySearch(arr, first, last, target);
      System.out.println("returnedValue=" + index);
    }
    return index;
  } // end binarySearch
} // end BinarySearch class
```

Do some work.

Then drill down.

Figure 11.8 Implementation of recursive binary search algorithm



```
/*******************************************************************
* BinarySearchDriver.java
* Dean & Dean
*
* This drives the BinarySearch class.
*******************************************************************/

public class BinarySearchDriver
{
  public static void main(String[] args)
  {
    int[] array = new int[] {-7, 3, 5, 8, 12, 16, 23, 33, 55};

    System.out.println(BinarySearch.binarySearch(
      array, 0, (array.length - 1), 23));
    System.out.println(BinarySearch.binarySearch(
      array, 0, (array.length - 1), 4));
  } // end main
} // end BinarySearchDriver class
```

Output:

```
first=0, last=8
first=5, last=8
first=5, last=6
first=6, last=6
found
returnedValue=6
returnedValue=6
returnedValue=6
6
first=0, last=8
first=0, last=4
first=0, last=2
first=2, last=2
not found
returnedValue=-1
returnedValue=-1
returnedValue=-1
-1
```

Reduce range and drill down.

Just return the answer.

Figure 11.9 Driver for the BinarySearch class in Figure 11.8

In Figure 11.9's output section, the shaded areas are outputs generated by the shaded print statements in Figure 11.8. In this recursion, the real work is done while the process is drilling down to the stopping condition. Notice how the first and last values converge on the match or the place where the match will be if it is there. The answer is generated at the point when the stopping condition is reached. The nested returns just pass this answer back. When you remove the shaded print statements from Figure 11.8, the only output that you will see is the unshaded parts of the output.

In a recursive binary search, as you can see, we simplify the problem by dividing the array into two nearly equally sized arrays. Then we continue dividing until each half contains no more than one element, which is the stopping condition.

Why is a binary search faster than a sequential search? The number of steps required for a sequential search equals *array*.length, whereas the number of steps required in a binary search equals only $\log_2$(*array*.length). For example, if there are 1 million items in the array, that's 1 million steps for a sequential search, but only about 20 steps for a binary search. Even if a typical binary-search step is more complicated than a typical sequential-search step, the binary search will still be significantly faster when the array is very long.

As we said in Section 9.7, a precondition for a binary search is that the data searched must be sorted. Section 9.8 provided an implementation for the relatively simple selection sorting algorithm. You could use this to prepare data for either the previous iterative binary search or the present recursive binary search. However, in Section 11.7, we'll show you a more efficient merge sorting algorithm, and that would be a better choice.

11.7 Merge Sort

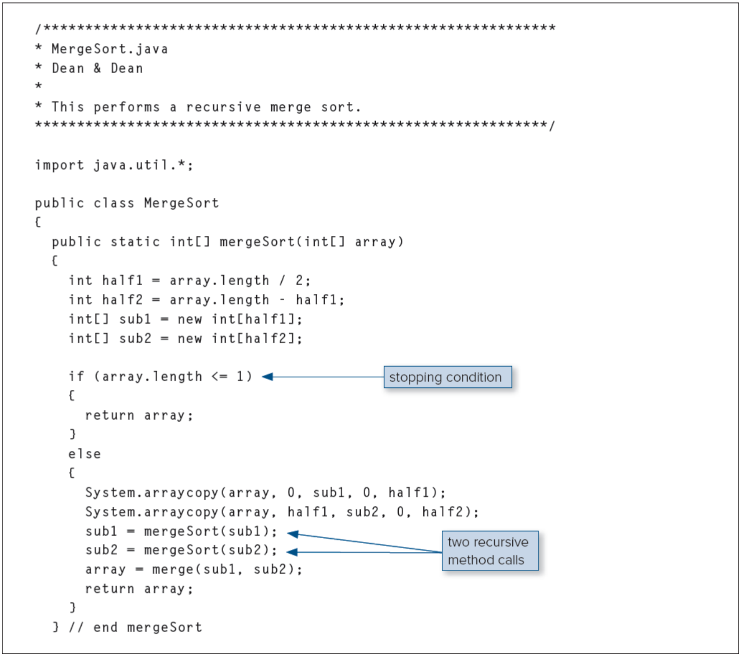
The binary search discussed in the previous section is a relatively efficient way to find something in a large collection of data. But for a binary search to work, the data must have been sorted already. In Section 9.8 of - Chapter 9, we described the relatively simple "selection sort" technique. Selection sort is good for small collections of data, but it's inefficient for large collections of data. Because the binary search you studied in the previous section works well for large collections of data, to prepare for that binary search, you'll want to use a sorting method that is also good for large collections of data. Merge sort is a good option.

The basic strategy in merge sorting is familiar—divide and conquer. In this respect, a merge sort is like a binary search. But this time, instead of making a recursive call for just one of the two halves, it makes recursive calls for both of them. In each of these recursive calls, it divides the current part in half, makes recursive calls for both smaller parts, and so forth, until a part has only one element. That's the stopping condition for the recursive branch that leads to that part.

Return sequences put things back together by merging parts, two at a time. Each return step merges all the elements in one part with all the elements in another part to form a larger part, until eventually, everything is back together in a single whole.

Figure 11.10a shows the recursive mergeSort method. The parameter is an unsorted array, called array. Local variable declarations create two subordinate arrays, sub1 and sub2. If array's length is an even number, the lengths of sub1 and sub2 are both equal to half the length of array. If array's length is an odd number, sub2 is one element longer than sub1. The stopping condition stops a recursion whenever array's length becomes one or zero. Negative array lengths are impossible. Whenever the stopping condition occurs, the returned array is the same as the parameter array.

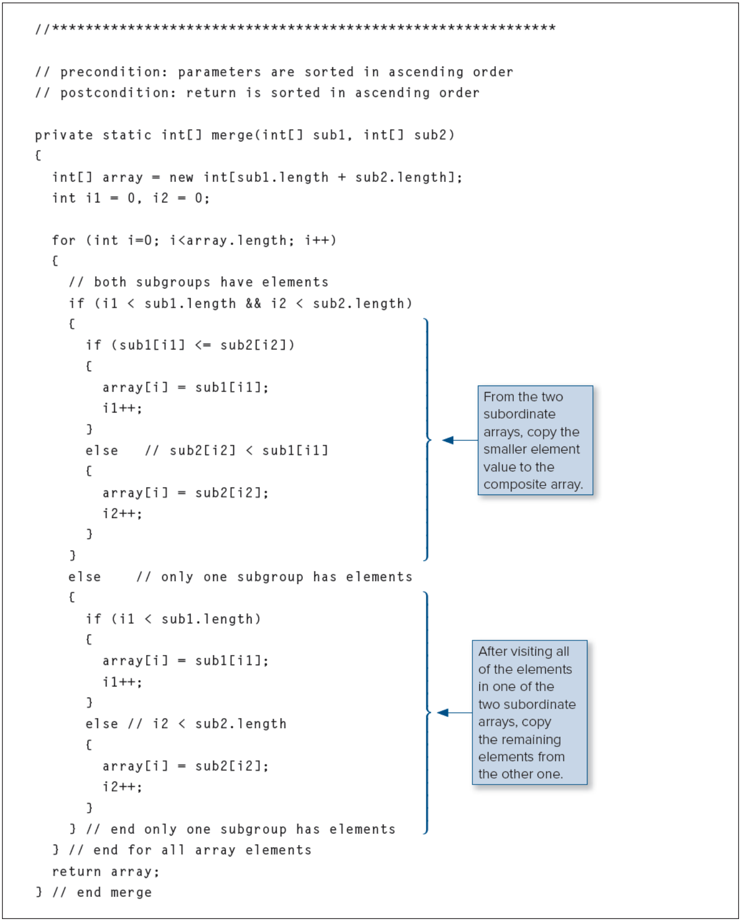
If the parameter array's length is two or more, execution falls to the "else" part of the "if, else" statement. Within the "else" clause, a pair of calls to System's arrayCopy method fills the two half-length subordinate arrays, sub1 and sub2, with the values in the lower and upper halves of array, respectively. (You learned about the arrayCopy method in Section 9.5.) Then, the mergeSort method recursively calls itself two times—once to sort the sub1 array and once to sort the sub2 array. After the two recursive calls, the mergeSort method calls the merge helper method. The merge method copies the sub1 and sub2 array elements in sorted order into a new array whose length equals the sum of the lengths of sub1 and sub2. The mergeSort method then returns the new array, which contains the same elements as the original array, but in sorted order.



```
/****************************************************************
* MergeSort.java
* Dean & Dean
*
* This performs a recursive merge sort.
****************************************************************/

import java.util.*;

public class MergeSort
{
  public static int[] mergeSort(int[] array)
  {
    int half1 = array.length / 2;
    int half2 = array.length - half1;
    int[] sub1 = new int[half1];
    int[] sub2 = new int[half2];

    if (array.length <= 1)
    {
      return array;
    }
    else
    {
      System.arraycopy(array, 0, sub1, 0, half1);
      System.arraycopy(array, half1, sub2, 0, half2);
      sub1 = mergeSort(sub1);
      sub2 = mergeSort(sub2);
      array = merge(sub1, sub2);
      return array;
    }
  } // end mergeSort
```

Figure 11.10a MergeSort program—part A This shows the mergeSort method, which calls itself recursively.

Take a look at the merge method in Figure 11.10b. It merges its two array parameters, sub1 and sub2, by copying the values from sub1 and sub2 into a new composite array such that the new array's values are sorted in ascending order. This process is made easier by the fact that when merge receives sub1, its elements are already in ascending order. The same is true for sub2. As you can see in Figure 11.10b, the merge method loops through the elements in sub1 and sub2, and for each loop iteration, it copies the smaller of the two elements (from sub1 or sub2) to the new composite array. As that loop iteration comparison - process continues, the composite array fills up with smaller values first and larger values last. After all of the elements from one of the subordinate arrays have been copied to the composite array, the loop stops and all of the remaining elements from the other subordinate array are copied to the unfilled higher-indexed elements in the composite array. The merge method then returns the composite array to mergeSort.



```
//************************************************************

// precondition: parameters are sorted in ascending order
// postcondition: return is sorted in ascending order

private static int[] merge(int[] sub1, int[] sub2)
{
  int[] array = new int[sub1.length + sub2.length];
  int i1 = 0, i2 = 0;

  for (int i=0; i<array.length; i++)
  {
    // both subgroups have elements
    if (i1 < sub1.length && i2 < sub2.length)
    {
      if (sub1[i1] <= sub2[i2])
      {
        array[i] = sub1[i1];
        i1++;
      }
      else   // sub2[i2] < sub1[i1]
      {
        array[i] = sub2[i2];
        i2++;
      }
    }
    else    // only one subgroup has elements
    {
      if (i1 < sub1.length)
      {
        array[i] = sub1[i1];
        i1++;
      }
      else // i2 < sub2.length
      {
        array[i] = sub2[i2];
        i2++;
      }
    } // end only one subgroup has elements
  } // end for all array elements
  return array;
} // end merge
```

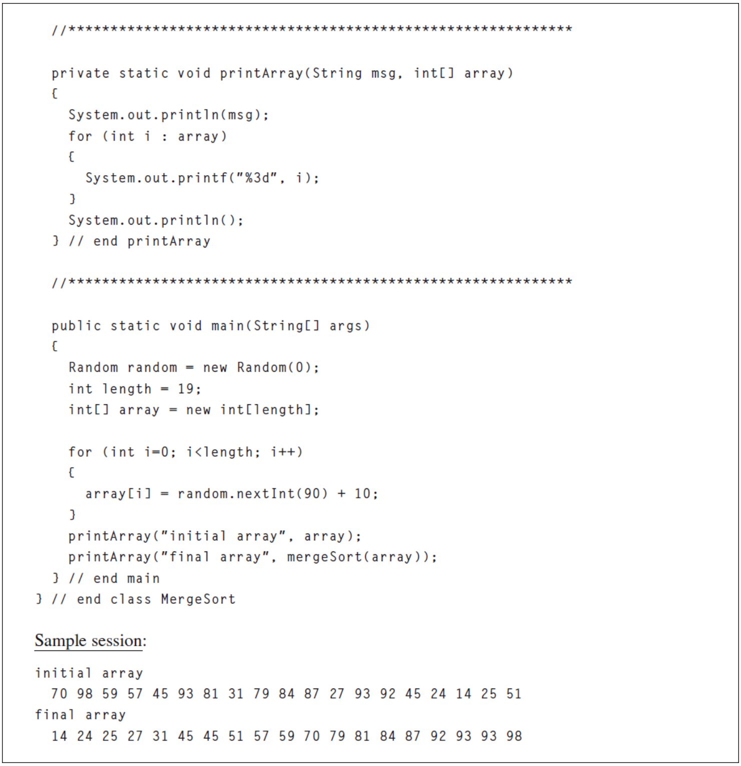
From the two subordinate arrays, copy the smaller element value to the composite array.

After visiting all of the elements in one of the two subordinate arrays, copy the remaining elements from the other one.

Figure 11.10b MergeSort program—part B This shows the merge method, which

merges two sorted parts into one sorted whole.

Now look at Figure 11.10c. It contains the rest of the MergeSort program—the main method and a helper method named printArray. The printArray method prints the contents of its passed-in array parameter. Also, as shown in Figure 11.10c’s sample session, it prints a description of the array, using the passed-in msg parameter for the description. Having this method available makes it easy to insert temporary diagnostic print statements anywhere in the code while the program is being developed and debugged.



```
  //***************************************************************

  private static void printArray(String msg, int[] array)
  {
    System.out.println(msg);
    for (int i : array)
    {
      System.out.printf("%3d", i);
    }
    System.out.println();
  } // end printArray

  //***************************************************************

  public static void main(String[] args)
  {
    Random random = new Random(0);
    int length = 19;
    int[] array = new int[length];

    for (int i=0; i<length; i++)
    {
      array[i] = random.nextInt(90) + 10;
    }
    printArray("initial array", array);
    printArray("final array", mergeSort(array));
  } // end main
} // end class MergeSort
```

Sample session:

```
initial array
  70 98 59 57 45 93 81 31 79 84 87 27 93 92 45 24 14 25 51
final array
  14 24 25 27 31 45 45 51 57 59 70 79 81 84 87 92 93 93 98
```

Figure 11.10c MergeSort program—part C This shows the program's driver and a display helper method.

The MergeSort program's main method employs a Random object to create a sequence of random numbers that are reproducible from one run to the next. Being able to reproduce random numbers is very helpful during testing and debugging. If you are not already familiar with this capability, you can read more about it in Section 5.8. Speaking of testing, we tested the MergeSort program with arrays of various sizes. You can see an array length value of 19 in Figure 11.10c. We also tested with an array whose length value was an even number and an array whose length value was zero. All the tests gave good results. Likewise, when you write your own programs, you should take the time to thoroughly test your programs with a wide variety of data.

11.8 Towers of Hanoi

In 1883, a French mathematician named Édouard Lucas proposed a puzzle based on an ancient legend. According to one version of this legend, a room in a temple in the city of Hanoi contains 64 golden disks, each with a different diameter and each with a hole in its center. The disks are stacked on three towering posts. At the beginning of the world, all the disks were stacked on just one of the posts, with the largest-diameter disk on the bottom and progressively smaller-diameter disks placed on top of each other. Figure 11.11 shows a simplified version of the problem, with only 4 disks instead of 64.

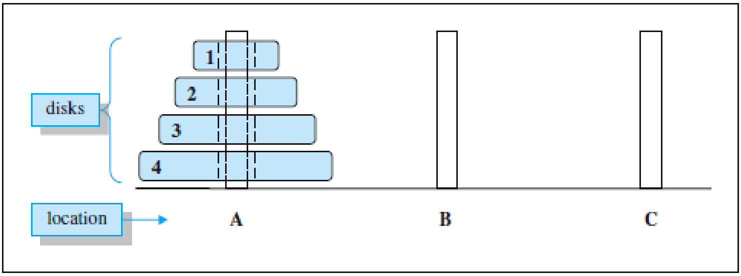


Figure 11.11 Setup for the Towers of Hanoi problem

The temple's monks have a backbreaking, mind-numbing mission. They are supposed to transfer all the disks from post A (the original post) to post C, in accordance with these commandments: (1) Move one disk at a time from one post to another post, (2) Never place any disk on top of a smaller disk. The legend says that when they finish this mission, they will be released from their labor, and the world will end.

Your task is to help the monks obtain the earliest possible release from their labor by giving them a computer program that specifies the optimum transfer sequence. The trick is to find a simple algorithm.

If you attempt to implement your solution with loops, you'll probably create a mess. But if you use recursion, you'll be more likely to succeed. Whenever you want to make a move, you need to identify a source location, s, a destination location, d, and a temporary location, t. For our overall goal, s is A, d is C, and t is B. As you progress toward the final solution, you'll have subordinate goals, with different locations for s, d, and t. Here is the general algorithm. It applies to any subset of disks from disk n down to disk 1, where n is any number from the maximum number of disks down to 1:

- Move the stack of disks above disk n from s to t.
- Move disk n to d.
- Move the stack of disks previously above disk n from t to d.

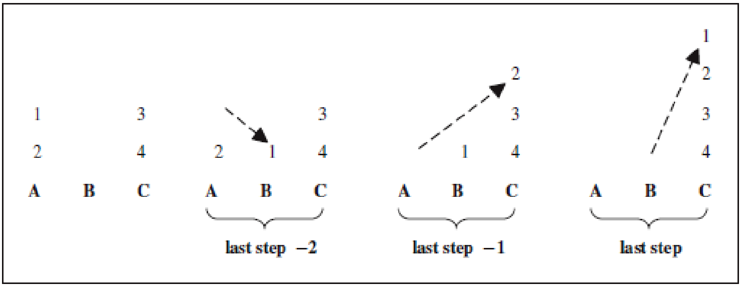
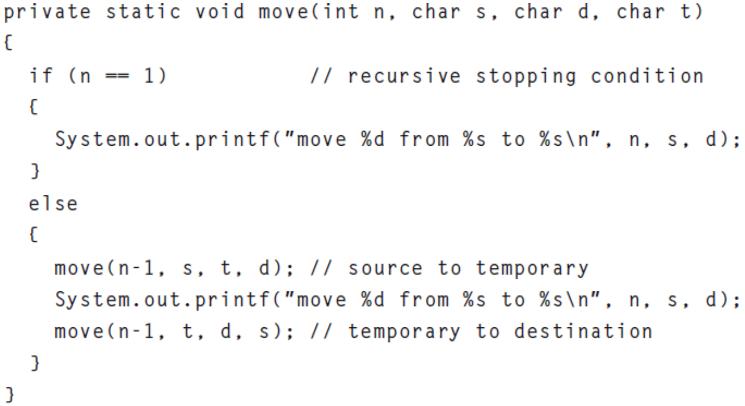

1
2
A
B
3
4
C
2
1
3
4
A
B
C
last step −2
2
3
1
4
A
B
C
last step −1
1
2
3
4
A
B
C
last step

Figure 11.12 Illustration of the Towers of Hanoi algorithm in action

Figure 11.12 shows an example of this algorithm in action. (This particular example happens to be the last few steps in the final solution.) The configuration on the left is a condition that exists shortly before the goal is attained. The configuration on the right is the final condition. The dashed arrows indicate each of the three operations described above for the simplest nontrivial case where there is only one disk above disk 2. The trivial case is the stopping condition. It's when you move the top disk, disk 1, from a source location to a destination location. An example of this appears as the last step in Figure 11.12. This happens to be the final stopping condition, but as you'll see, a program that solves the Towers of Hanoi problem will hit the stopping condition and automatically restart many times during its execution.

Assume that the evaluation process is currently in the left frame of Figure 11.12. The next operation calls the following move method with arguments (2, 'A', 'C', 'B'). Within this method, the else clause's first subordinate move method call uses arguments (1, ‘A’, ‘B’, ‘C’) to implement Figure 11.12’s “last step-2.” The subsequent printf statement implements Figure 11.12 “last step −1.” The else clause’s second subordinate move method call uses arguments (1, ‘B’, ‘C’, ‘A’) to implement Figure 11.12’s “last step.”

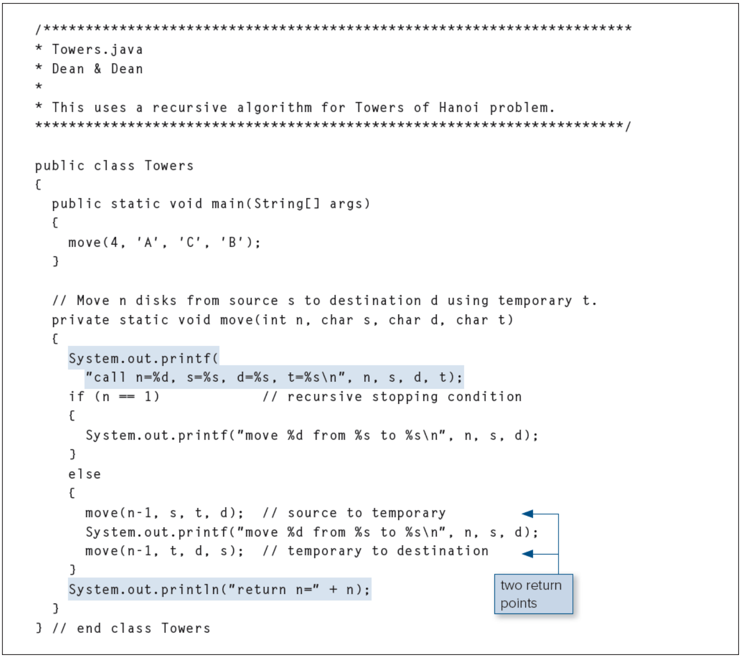


```
private static void move(int n, char s, char d, char t)
{
  if (n == 1)           // recursive stopping condition
  {
    System.out.printf("move %d from %s to %s\n", n, s, d);
  }
  else
  {
    move(n-1, s, t, d); // source to temporary
    System.out.printf("move %d from %s to %s\n", n, s, d);
    move(n-1, t, d, s); // temporary to destination
  }
}
```

The initial call to the recursive method should establish the overall goal, which is to move the entire tower from location A to location C. To get the largest disk to the new location first, you should start with the maximum possible n. The algorithm says you can do this by moving the subordinate set of disks, 1, 2, and 3 from the source location, A, to the temporary location, B. Then you move disk 4 from the source location, A, to the destination location C. Then you move the subordinate set of disks 1, 2, and 3 from the temporary location, B, to the destination location, C, thereby putting them on top of the largest disk, 4. The problem with this process is that the rules don’t permit moving more than one disk at a time. So, to move the subordinate set of disks, 1, 2, and 3, you must call the same method recursively to move just disks 1 and 2. To do that, you must call the same method recursively again to move just disk 1.

Of course, the first disk to move is disk 1, but it’s hard to know where to put it. Should you move it to location B or to location C? The purpose of the program is to tell you exactly how to proceed. The program is displayed in Figure 11.13. The shaded print statements are not part of the solution, and they should be omitted from a finished product. We inserted them just to help you trace the torturous recursive activity—if you want to. For each method invocation, they print right after the method is called and just before it returns to show you the details of what's happening.

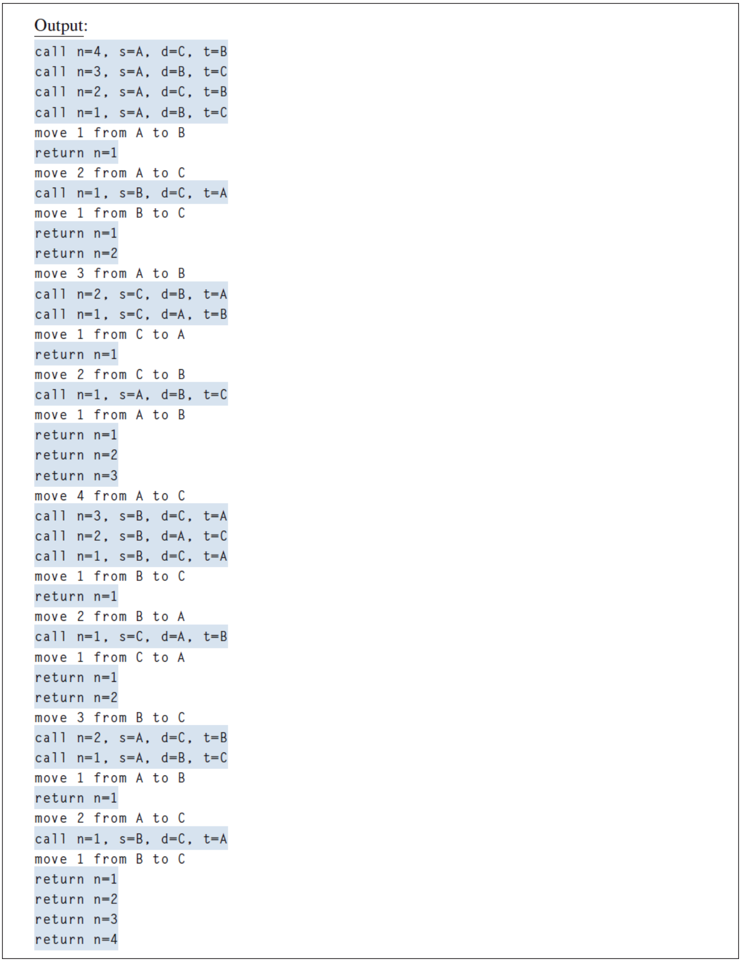
Figure 11.14 displays the output. The shaded lines are lines printed by the shaded print statements in Figure 11.13. As we said, they are just to help you see what happened and are not part of the solution. The solution is given by the unshaded outputs. The most difficult part of tracing a recursive algorithm like this is keeping track of the place from which a call was made, and therefore where execution resumes after a return. As our note in Figure 11.13 indicates, sometimes the call is made from the “source to temporary” statement, and sometimes the call is made from the “temporary to destination” statement. Fortunately, if you define a recursive algorithm correctly, you can ignore the details of how it plays out during program execution.



```
/**********************************************************************
* Towers.java
* Dean & Dean
*
* This uses a recursive algorithm for Towers of Hanoi problem.
**********************************************************************/

public class Towers
{
  public static void main(String[] args)
  {
    move(4, 'A', 'C', 'B');
  }

  // Move n disks from source s to destination d using temporary t.
  private static void move(int n, char s, char d, char t)
  {
    System.out.printf(
      "call n=%d, s=%s, d=%s, t=%s\n", n, s, d, t);
    if (n == 1)            // recursive stopping condition
    {
      System.out.printf("move %d from %s to %s\n", n, s, d);
    }
    else
    {
      move(n-1, s, t, d);  // source to temporary
      System.out.printf("move %d from %s to %s\n", n, s, d);
      move(n-1, t, d, s);  // temporary to destination
    }
    System.out.println("return n=" + n);
  }
} // end class Towers
```

two return points

Figure 11.13 Solution to the Towers of Hanoi problem Shaded statements are for diagnostic tracing. Remove them for final implementation.



Output:

```
call n=4, s=A, d=C, t=B
call n=3, s=A, d=B, t=C
call n=2, s=A, d=C, t=B
call n=1, s=A, d=B, t=C
move 1 from A to B
return n=1
move 2 from A to C
call n=1, s=B, d=C, t=A
move 1 from B to C
return n=1
return n=2
move 3 from A to B
call n=2, s=C, d=B, t=A
call n=1, s=C, d=A, t=B
move 1 from C to A
return n=1
move 2 from C to B
call n=1, s=A, d=B, t=C
move 1 from A to B
return n=1
return n=2
return n=3
move 4 from A to C
call n=3, s=B, d=C, t=A
call n=2, s=B, d=A, t=C
call n=1, s=B, d=C, t=A
move 1 from B to C
return n=1
move 2 from B to A
call n=1, s=C, d=A, t=B
move 1 from C to A
return n=1
return n=2
move 3 from B to C
call n=2, s=A, d=C, t=B
call n=1, s=A, d=B, t=C
move 1 from A to B
return n=1
move 2 from A to C
call n=1, s=B, d=C, t=A
move 1 from B to C
return n=1
return n=2
return n=3
return n=4
```

Figure 11.14 Output from the Towers program in Figure 11.13 Unshaded statements are the solution. Shaded ones provide a trace.

We recommend that you cut out four little paper disks of different sizes, number them 1 through 4 from smallest to largest, and build a tower at location “A” on your left. Then move the disks one at a time as the unshaded outputs in Figure 11.14 say. You’ll see that the tower does in fact get moved from location “A” to location “C” in precise conformity with the specified rules. This works because the eventual moves are informed by goal information in earlier method calls.

Figure 11.14 shows that the monks would need a minimum of 15 moves to complete the task if there were just four disks. The number 64 is 16 times larger than 4. So if the minimum number of moves increased linearly with the number of disks, the monks would need a minimum of $16 \times 15 = 240$ moves to complete the task and bring about the end of the world. At the time of this book’s writing, the world still exists, so perhaps there’s something wrong with the calculation? Perhaps the number of moves increases more precipitously as the number of disks increases?

Wikipedia says with 64 disks, the minimum number of moves is $(2^{64} - 1)$, which is 18,446,744,073, 709,551,615 moves. If the monks could make one move per second on a 24/7 basis, they would need a total of 585 billion years to complete the task. Whew! That's a relief. No need to rouse the Four Horsemen of the Apocalypse just yet. But is Wikipedia's formula correct? Can you think of a way to test the validity of that formula? One of the end-of chapter exercises asks you to check that formula with computer simulation and inference.

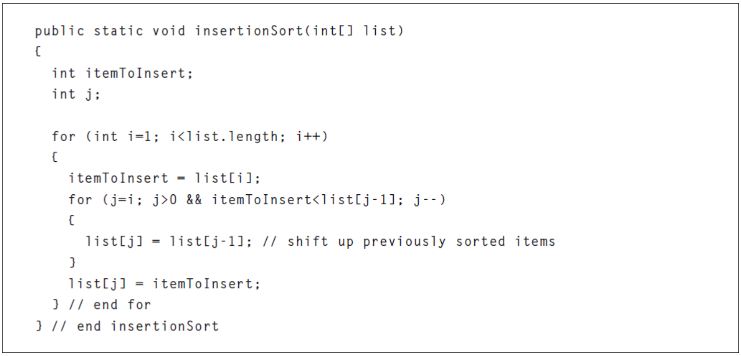
11.9 Problem Solving with Performance

Analysis

Performance is another word for efficiency. Efficiency is important because if a proposed implementation does not execute in the available time or the available computer memory, it's not useful. Many applications are time critical. For example, if you speed up the rate at which the trees grow in the following section's program, eventually that program crashes. Frequently, there are several different ways to solve a problem. The performance estimation techniques reviewed and introduced in this section will help you decide which way is best.

We quantify the performance of an algorithm by measuring or calculating the execution time or the space (memory) required. High performance means short time and minimal space. Time and space requirements are partially correlated. Usually, when analyzing an algorithm, more attention is paid to time analysis than space analysis, and we follow that pattern in this section. Specifically, we'll focus on estimating the number of computational steps required to complete the algorithm as a function of data. Knowing an algorithm's estimated number of computational steps is a critical part of time complexity analysis (*time complexity analysis* is the formal term for the study of an algorithm's execution speed).

Since Chapter 4, you've seen many examples of loops, and a typical for loop header usually tells you the maximum number of iterations. For example, if the loop iterates through an array, the number of iterations equals the array's length. If each iteration takes the same amount of time, the time needed to iterate through the array increases linearly with array length. Sometimes a loop includes another loop nested inside it. Then the total number of iterations is the number of iterations in the outer loop times the number of iterations of the inner loop. The code in Figure 11.15 illustrates this as it implements an algorithm called *insertion sort*. Some people use this instead of selection sort to sort a hand of cards.



```
public static void insertionSort(int[] list)
{
  int itemToInsert;
  int j;

  for (int i=1; i<list.length; i++)
  {
    itemToInsert = list[i];
    for (j=i; j>0 && itemToInsert<list[j-1]; j--)
    {
      list[j] = list[j-1]; // shift up previously sorted items
    }
    list[j] = itemToInsert;
  } // end for
} // end insertionSort
```

Figure 11.15 Insertion sort method

In the insertion sort method, note how the inner for loop starts at the outer loop’s current index and iterates down through previously sorted items, shifting them upward as the iteration proceeds, until the item to insert is less than a previously sorted item. Then it inserts the new item there. The portion of the array that is already sorted grows as the outer for loop progresses.

If the array is already sorted, the condition, itemToInsert<list[j-1] is always false, and the inner loop never executes. In this best case, the total number of steps is just the number of steps through the outer loop. Because the number of iterations in the outer for loop is list.length 1, in this best case, the total number of steps is

<*minimum steps*> = list.length - 1

If the array is initially in reverse order, the condition itemToInsert<list[j-1] is always true, and the inner loop always shifts up all previously sorted items. In this worst case, the total number of steps is

<*maximum steps*> = (list.length - 1) * list.length / 2

If the array is initially in random order, on average, the inner loop shifts up about half of the previously sorted items. In this average case, the total number of steps is

<*average steps*> = (list.length - 1) * list.length / 4

The analysis above shows that performance depends not only on the nature of the algorithm, but also on the state of the data. The experimental measurements and discussion in Section 10.8 identify another confounding factor. The time needed by computer hardware to perform a particular operation is a function of the number of times it performs that operation. For example, we expect the time to get or set an ArrayList element to be independent of list length. In contrast, measured results presented by and associated with Figure 10.8 show that as an ArrayList's length increases from 100 to 1,000 to 10,000, the average get and set time decreases from 401 ns to 156 ns to 74 ns. Also, we expect that the time to get or set a linked list element to increase linearly with list length. The measured results show that as the length of a LinkedList increases from 100 to 1,000 to 10,000, the average get and set time increases from 1,248 ns to 1,861 ns to 8,597 ns. There is a general increase, but it is not exactly linear.

These hard-to-predict performance variations suggest that we should not get too carried away with precision when we do performance analysis. To model performance, we could try to approximate the total time or space required to perform a task with something like this:

$$<\textit{time or space required}> \approx a * f(n) + b$$

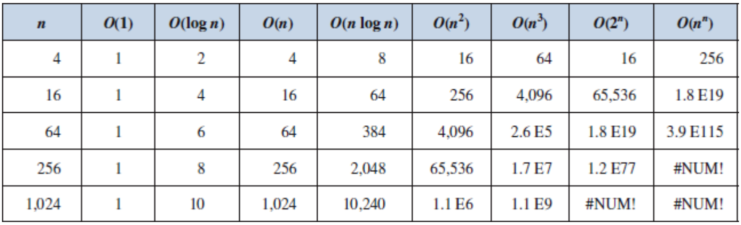
In this formula, n is the total number of elements, and f(n) means “function of n.” a is a scaling factor that indicates the time or space required for each n. b is a constant that might represent a fixed setup time or a common working space. We might, for example, use b to model the unexpectedly high measured get and set times for the shortest lists described in the previous paragraph. But usually, we elect to simplify the formula by dropping a and b, and writing:

$$<\text{time or space required}> \approx O(n)$$

The right side of this equation is what computer scientists call *Big O* notation. The *O* in this term stands for “order of magnitude.” A Big O function tells us approximately how time and/or space requirements grow as n (the size of the algorithm’s input data) grows. Algorithms with smaller Big O functions tend to run faster and/or use less memory. Algorithms with larger Big O functions tend to run slower and/or use more memory. If the time or space required is approximately the same for all n, we say “the problem is of order one,” or more simply, “it’s $O(1)$,” where “O” is the letter, not the number zero.

As n increases, if f(n) approaches a linear dependence on n, we say, “The problem is of order *n*,” or more simply, “It’s $O(n)$.” If f(n) approaches a parabolic dependence on n, we say, “it’s $O(n^2)$.” If f(n) approaches a cubic dependence on n, we say, “it’s $O(n^3)$.” And it continues like that. When f(n) approaches a logarithmic dependence on n, we say, “It’s $O(\log n)$.” When f(n) approaches a dependence like n * log(n), we say, “It’s $O(n \log n)$.” For some really complex problems, the growth rate might increase exponentially, and we say, “It’s $O(2^n)$,” or perhaps, “It’s $O(n^n)$.”

Table 11.1 shows these Big O growth rates and how they affect several values of n. For the logarithms, we use base 2. The #NUM! symbol means the computed number is larger than our computer could hold. Clearly, the growth rates for $O(1)$ and $O(\log(n))$ are slow, whereas the growth rates for $O(2^n)$ and $O(n^n)$ are fast—so fast that computer scientists call exponential dependences *intractable*. *Intractable* means that for large *n*, it's virtually impossible to obtain exact answers.



n	$O(1)$	$O(\log n)$	$O(n)$	$O(n \log n)$	$O(n^2)$	$O(n^3)$	$O(2^n)$	$O(n^n)$
4	1	2	4	8	16	64	16	256
16	1	4	16	64	256	4,096	65,536	1.8 E19
64	1	6	64	384	4,096	2.6 E5	1.8 E19	3.9 E115
256	1	8	256	2,048	65,536	1.7 E7	1.2 E77	#NUM!
1,024	1	10	1,024	10,240	1.1 E6	1.1 E9	#NUM!	#NUM!

Table 11.1 Big O growth rates and how they affect several values of n Logarithms are base 2, and #NUM! means the number is larger than our computer could hold.

Compared to performing a detailed analysis, it's relatively easy to determine Big O performance. Here are some examples, taken from material in Chapters 9, 10, and 11:

- [§9.7] Sequential search: $O(n)$
- [§9.7] Binary search: $O(\log n)$
- [§9.8] Selection sort: $O(n^2)$
- [§9.9] Two-dimensional array fill: $O(n^2)$
- [§10.2] List's contains method: $O(n)$
- [§10.7] ArrayList's get and set methods: $O(1)$
- [§10.7] LinkedList's get and set methods: $O(n)$
- [§10.7] List's indexed remove and add methods: $O(n)$
- [§10.9] ArrayDeque's remove and add methods: $O(1)$
- [§10.10] HashMap's put, get, contains, and remove methods: $O(1)$
- [§10.10] TreeSet's add and get methods: $O(\log n)$

Except for the additional burden of method-call overhead, the performance of a recursive calculation is like that of an iterative counterpart:

- [§11.4] Factorial: $O(n)$
- [§11.4] PrintReverseMessage: $O(n)$
- [§11.6] Binary search: $O(\log n)$
- [§11.7] Merge sort: $O(n \log n)$
- [§11.8] Towers of Hanoi: $O(2^n)$
- [§11.9] drawBranches: $O(n)$, where n is the number of branches
- [§11.10] Insertion sort: $O(n^2)$, or $O(n)$ if already sorted or nearly sorted

11.10 GUI Track: Drawing Trees with a Fractal Algorithm (Optional)

In our discussion of software engineering in Section 8.7, we identified a particular design pattern called MVC, which stands for Model-View-Controller. This section presents a program that has MVC organization, with a separate class for each of the three concerns: model, view, and controller.

Model

All of the native cells in a living tree contain exactly the same DNA. That identical DNA enables the cells to reproduce themselves and replicate their own attributes in other cells. As a living tree grows, its cells make decisions about when to start new branches and what angles those branches should take. Since cells throughout the tree have similar attributes, they tend to make the same decisions. Persistent environmental factors also affect those decisions. If one side of a tree gets more sun, it grows faster on that side. A strong prevailing wind influences the direction of growth. Such factors can cause repeated asymmetry.

Computer recursion is an analog of the recurrence of similar patterns in an organism. So, it’s natural for us to use recursion to describe that organism. Our tree simulation will employ recursion to describe growth in time and extension in space. We’ll model growth in time like we computed a factorial in Section 11.3. A method that models growth in one year calls itself for growth in the previous year and so forth, back to the year of its planting. Then annual growths accumulate as the recursive calls return. For the growth in any one year we’ll use a variation of the *logistic* equation employed in Section 6.13.

At any given time, we’ll model extension in space with binary recursion of a simple geometrical pattern—a straight section followed by a fork with two branches. The left branch will go 30 degrees to the left with a length equal to 75% of the length of the straight section. The right branch will go 50 degrees to the right with a length equal to 67% of the length of the straight section. Each recursive branch stops when its length becomes equal to or less than a certain minimum length. An object created by repeating a pattern at different scales is called a *fractal*. A mathematical fractal displays *self-similarity* on all scales.

Self-similarity exists only between limits. A tree grows in time from the size of an initial shoot to some maximum size. At any given time, a tree's components range in size from the size of the trunk to the size of the smallest twig. These limits will establish our recursive starting and stopping conditions.

Figure 11.16 models a botanical tree

in a class call TreeModel. It imports the Java API Line class to represent a tree's trunk and branches. ANNUAL_GROWTH_RATE specifies initial annual growth rate, and MAX_TRUNK_LENGTH limits tree size. The getTrunkLength method uses our logistic growth formula to model growth in trunk length from the year of a tree's planting to the current year. The parameter that varies from one call to the next is the tree's age. As in Section 11.3's recursive factorial method, this parameter starts at a highest value (current year minus year of planting) and decrements[2] to a stopping condition (age = = 0).

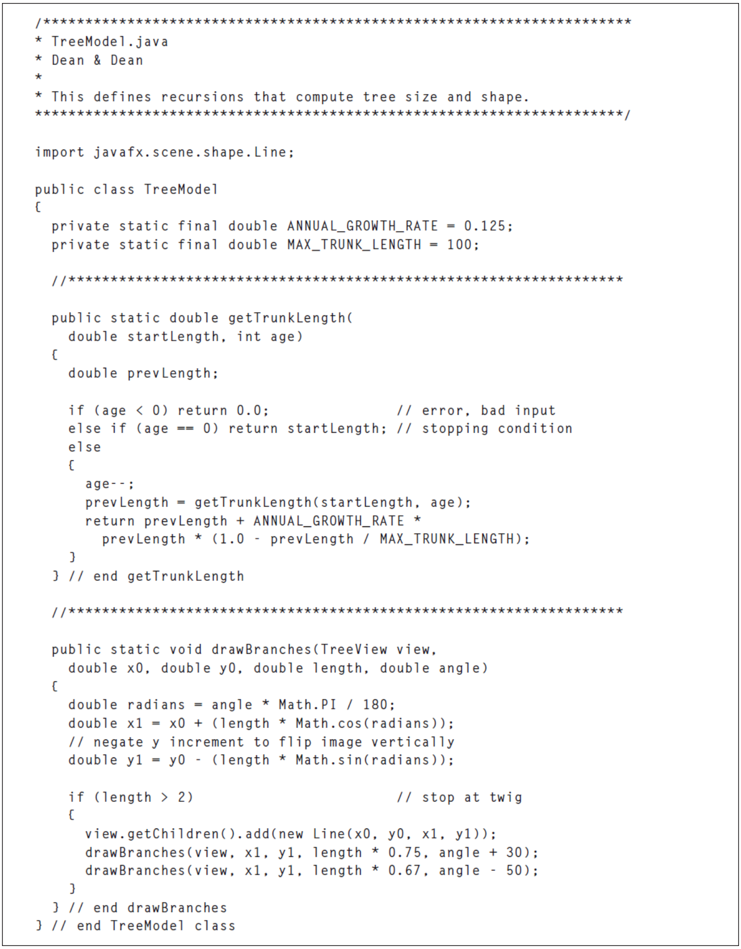
Now look at the drawBranches method. The first argument is a reference to the TreeView class that will have called this method. This invariant common reference makes it easy for all recursive manifestations of this method to send graphical information back to the original caller. The next four arguments are the starting location, length, and angle of the current branch. These arguments change as recursion progresses.

The first declaration in the drawBranches method converts degrees to radians. The second declaration computes the x-position of the outer end of the current branch. The third declaration computes the y-position of the outer end of the current branch. Since the computer displays increasing y-pixels in the downward direction, we use a minus sign to make y1 appear above y0 on the screen when y1 is greater than y0.

The if condition in the drawBranches method makes the method return without doing anything if the length parameter is two or less. This is the recursive stopping condition. It corresponds to branches that are twigs. Within the if clause, the first statement draws the current branch. The next two statements recursively call the same method to draw the subordinate branches that fork to left and right, respectively. The left branch's length is 75% of the current branch's length, and the right branch's length is 67% of the right branch's length. So the branches keep getting shorter as the recursion proceeds from trunk to twigs.

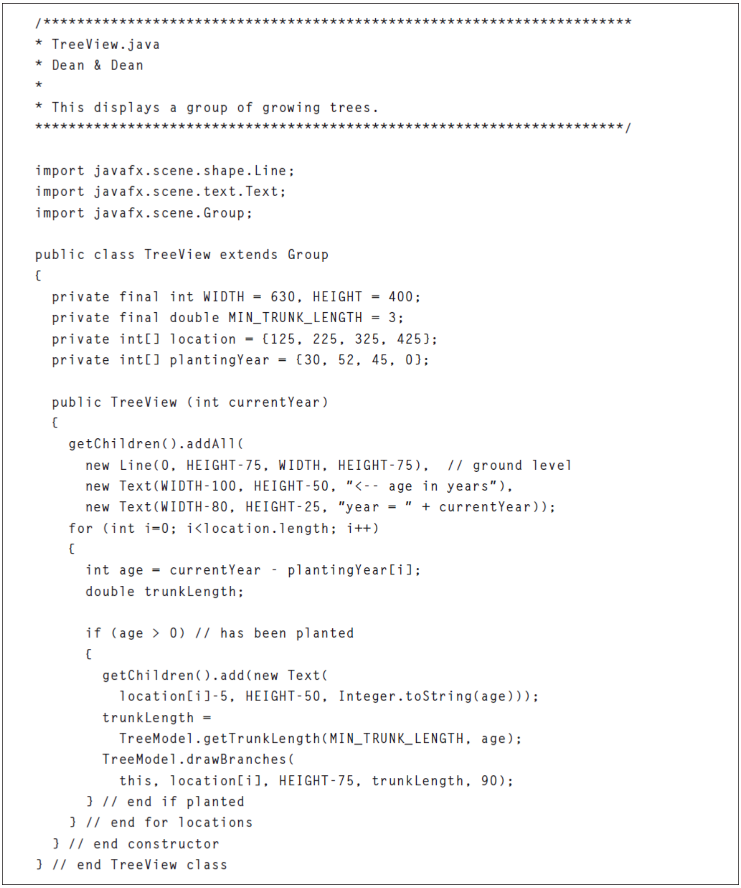
View

Figure 11.17 shows the TreeView class, which performs our program's display operations. This class imports the JavaFX Line and Text classes for the display's components. It imports the JavaFX Group class to contain those components. TreeView extends Group to acquire Group capabilities itself. This enables an instance of TreeView to be the *scene graph* that contains everything in a Stage's Scene.



```java
/**********************************************************************
* TreeModel.java
* Dean & Dean
*
* This defines recursions that compute tree size and shape.
**********************************************************************/

import javafx.scene.shape.Line;

public class TreeModel
{
  private static final double ANNUAL_GROWTH_RATE = 0.125;
  private static final double MAX_TRUNK_LENGTH = 100;

  //******************************************************************

  public static double getTrunkLength(
    double startLength, int age)
  {
    double prevLength;

    if (age < 0) return 0.0;               // error, bad input
    else if (age == 0) return startLength; // stopping condition
    else
    {
      age--;
      prevLength = getTrunkLength(startLength, age);
      return prevLength + ANNUAL_GROWTH_RATE *
        prevLength * (1.0 - prevLength / MAX_TRUNK_LENGTH);
    }
  } // end getTrunkLength

  //******************************************************************

  public static void drawBranches(TreeView view,
    double x0, double y0, double length, double angle)
  {
    double radians = angle * Math.PI / 180;
    double x1 = x0 + (length * Math.cos(radians));
    // negate y increment to flip image vertically
    double y1 = y0 - (length * Math.sin(radians));

    if (length > 2)                        // stop at twig
    {
      view.getChildren().add(new Line(x0, y0, x1, y1));
      drawBranches(view, x1, y1, length * 0.75, angle + 30);
      drawBranches(view, x1, y1, length * 0.67, angle - 50);
    }
  } // end drawBranches
} // end TreeModel class
```

Figure 11.16 The TreeModel class computes size and shape of a tree



```
/*************************************************************************
* TreeView.java
* Dean & Dean
*
* This displays a group of growing trees.
*************************************************************************/

import javafx.scene.shape.Line;
import javafx.scene.text.Text;
import javafx.scene.Group;

public class TreeView extends Group
{
  private final int WIDTH = 630, HEIGHT = 400;
  private final double MIN_TRUNK_LENGTH = 3;
  private int[] location = {125, 225, 325, 425};
  private int[] plantingYear = {30, 52, 45, 0};

  public TreeView (int currentYear)
  {
    getChildren().addAll(
      new Line(0, HEIGHT-75, WIDTH, HEIGHT-75),  // ground level
      new Text(WIDTH-100, HEIGHT-50, ”<-- age in years”),
      new Text(WIDTH-80, HEIGHT-25, ”year = ” + currentYear));
    for (int i=0; i<location.length; i++)
    {
      int age = currentYear - plantingYear[i];
      double trunkLength;

      if (age > 0) // has been planted
      {
        getChildren().add(new Text(
          location[i]-5, HEIGHT-50, Integer.toString(age)));
        trunkLength =
          TreeModel.getTrunkLength(MIN_TRUNK_LENGTH, age);
        TreeModel.drawBranches(
          this, location[i], HEIGHT-75, trunkLength, 90);
      } // end if planted
    } // end for locations
  } // end constructor
} // end TreeView class
```

Figure 11.17 The TreeView class displays a group of trees

The WIDTH and HEIGHT constants indirectly establish our scene graph's initial width and height and therefore the initial width and height of the scene and stage. They do this by establishing the left end and width of the largest initial horizontal component (a horizontal Line representing the surface of the earth) and the vertical position of the lowest component (a Text object identifying the current time).

MIN_TRUNK_LENGTH (like MAX_TRUNK_LENGTH) is logically a property of the TreeModel class, but we shifted it to the TreeView class to make it easier for a variation of this program to plant trees that are larger than seedlings. The location and plantingYear arrays establish the quantity and location of trees and their planting years in the current version of the program. But it would be relatively easy to add get and set methods that allowed a controller to specify different quantities, locations, and planting times by making different assignments to TreeView's location and plantingYear instance variables.

The currentYear parameter in

TreeView's constructor has the same starting year as the elements of the plantingYear. So the difference, (currentYear - plantingYear[i]) represents the age of the tree at location i.

The (extended Group) constructor creates a scene graph by adding children. First it adds three common children—a horizontal Line to represent the earth's surface and two Text items in the lower right corner of the display. Then it employs a for loop to step through the elements of the location array. The for loop's first statement declares and initializes the age of the tree at the current location. Its second statement declares a trunkLength variable.

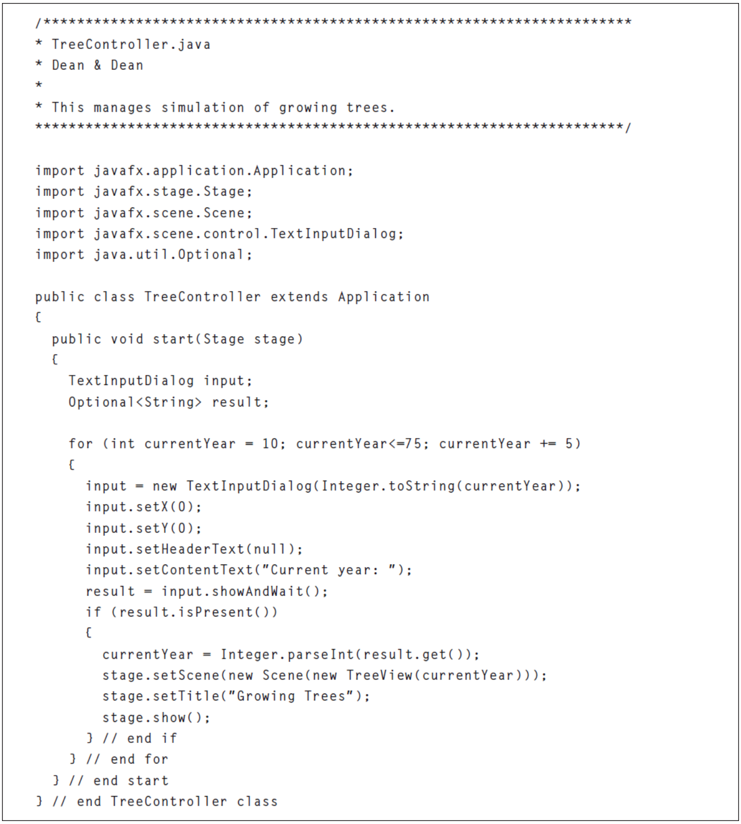
If a particular tree's age is greater than zero, it has been planted and has been growing for at least one year. So, we put Text under its location that displays its age. We call TreeModel's getTrunkLength method to determine its current trunk length. And we call TreeModel's drawBranches method to add a separate Line component for its trunk and for each of its (subordinate) branches to TreeView's list of children. An old tree will contribute a large number of such Line components a TreeView's list of children.

Controller

Figure 11.18 shows the TreeController class. This is the class that implements this program's GUI. Accordingly, it imports the JavaFX's Application, Stage, Scene, and TextInputDialog classes, and it imports Java's Optional class to capture results from a TextInputDialog. As usual, this class extends Application, and its only method is the obligatory overriding start method.

The start method contains a for loop that steps through currentYear values of 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, and 75. For each of these, it instantiates a TextInputDialog with that currentYear as the default input. To watch the group of trees grow, develop, and mature in a crude time-lapse "movie," press Enter or click OK 14 times. You'll see the succession as 14 windows appear one after another—each new one replacing the previous one. Alternatively, you can enter any particular year value to see a new window showing what the trees look like for that current year. As long as your year is not greater than 75, you can do this an indefinite number of times to see the condition at the end of any current year. If you enter greater than 75, you'll get a new window for that year, but the dialog will disappear, and execution will terminate when you close that final window. After any given tree gets to be about 80 years old, it will have reached nearly full size, and it won't grow much more.

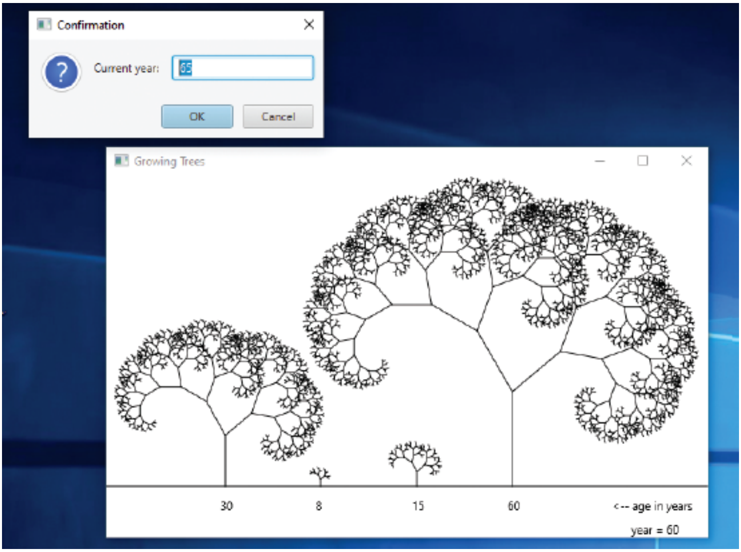
The code in Figure 11.18 works well as long as you just click OK or optionally enter an integer after "Current year" in the Confirmation dialog box before clicking OK. However, if you happen to enter a non-integer (or blank) after "Current year", subsequently clicking OK makes the program crash. Chapter 15 describes ways to avoid such crashing.



```
/************************************************************************
* TreeController.java
* Dean & Dean
*
* This manages simulation of growing trees.
************************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.TextInputDialog;
import java.util.Optional;

public class TreeController extends Application
{
  public void start(Stage stage)
  {
    TextInputDialog input;
    Optional<String> result;

    for (int currentYear = 10; currentYear<=75; currentYear += 5)
    {
      input = new TextInputDialog(Integer.toString(currentYear));
      input.setX(0);
      input.setY(0);
      input.setHeaderText(null);
      input.setContentText("Current year: ");
      result = input.showAndWait();
      if (result.isPresent())
      {
        currentYear = Integer.parseInt(result.get());
        stage.setScene(new Scene(new TreeView(currentYear)));
        stage.setTitle("Growing Trees");
        stage.show();
      } // end if
    } // end for
  } // end start
} // end TreeController class
```

Figure 11.18 The TreeController class drives TreeView and TreeModel

Figure 11.19 shows what the displayed window looks like when the user accepts all the default current-year values and clicks OK 11 times in succession. The tree on the right is within about 5% of its maximum size.


Confirmation
OK
Cancel
Growing Trees
30
8
15
60
<-- age in years

Current year: 65

year = 60

Figure 11.19 View of simulated tree growth obtained after clicking OK six times

Summary

- A recursive method repeats a given pattern of behavior by calling itself until it reaches a stopping condition.
- We can describe the calculation of a factorial with the recurrence relation, n! = n * (n-1)!, and the

stopping condition, if (n = = 1 || n = = 0).

- The trace of a recursion restates the recurrence relation with changing variable values and question marks for unknowns until it reaches a stopping condition. Then it backtracks, substituting previously determined values for question marks until it returns from the first recursive call.
- If a method's body includes just one recursive call to itself, and the call comes at the end of the method's body, it's called *tail recursion.* It's relatively easy to convert a tail recursion to a loop calculation.
- Binary search is an efficient technique for searching an already ordered array. Binary search lends itself well to recursion, where the binary search is conducted on the

two halves of the original array.

- Merge sort is an efficient technique for sorting a large array. Merge sort lends itself well to recursion, where the merge sort is conducted on the two halves of the original array.
- The Towers of Hanoi puzzle starts with different-sized disks stacked at one location with smaller disks above larger disks. The task is to move all the disks to another location, one at a time, never placing any disk on top of a smaller disk, and using only one other location as a temporary repository. There is a recursive solution that solves this difficult problem in a relatively straightforward manner.
- Characterize time performance by estimating the number of computational steps required as a

function of data quantity. This often depends on the condition of the data, but for simplicity, we often ignore this and also ignore additive and multiplicative factors with Big O representation.

- To produce a GUI animation that simulates growing trees, you can partition the tasks into a model class, a view class, and controller class. To model each tree's growth, you can use a recursive fractal algorithm.

Review Questions

---

§11.2 Guidelines for Writing a Recursive Method

1. Fill in the blank: "A recursive method needs to have at least one parameter whose value ________ with successive methods calls.”

2. When a stopping condition is satisfied, a recursive method calls itself. (T / F)

§11.3 A Recursive Factorial Method

3. Write a sufficient stopping condition for the recursive calculation of a factorial.

4. In Figure 11.2’s factorial calculation, during the calling sequence (before reaching the stopping condition), what is the value of nF when n equals 2?

§11.4 Comparison of Recursive and Iterative Solutions

5. Fill in the blank: “If the method’s body includes two recursive calls to itself and the method executes both calls, the method is said to exhibit

________.”

6. If it’s just as easy to solve a problem with a loop as with recursion, which solution is preferable and why?

7. In the printReverseMessage method of Figure 11.7, what variable changes as the recursion progresses, and what is the stopping condition?

§11.5 Recursive Method Evaluation Practice

8. Write the recurrence relation for a factorial using the functional notation, f(x).

9. What’s wrong with the stopping condition in this specification?

$$f(x) = \begin{cases} f(x - 2) + x & x > 0 \\ x - 2 & x = 0 \end{cases}$$

§11.6 Binary Search

10. How many recursive method

calls does it take to binary search an array with a length of 1,000?

11. Because a recursive binary search uses tail recursion, it's easy to convert it to a loop solution. (T / F)

§11.7 Merge Sort

12. The recursive merge sort algorithm uses binary recursion. (T / F)

13. In a recursive merge sort, what is the recursive stopping condition?

14. In a merge sort, the next element to go into the merged array is the smaller of the smallest elements in the two subordinate arrays. What happens when one of the two subordinate arrays becomes empty?

§11.8 Towers of Hanoi

15. What is the algorithm used to move a stack of disks whose bottom disk is an arbitrary disk n?

16. Describe the arguments used for the first recursive method call in the algorithm in question 15 in terms of the current method's parameters.

§11.10 GUI Track: Drawing a Tree with a Fractal Algorithm (Optional)

17. What is a fractal?

18. In our tree-drawing program, what does the model do?

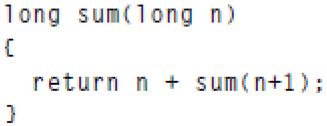
19. In our tree-drawing program, what does the view do?

20. In our tree-drawing program, what does the controller do?

Exercises

1. [after §11.2] The text says: “The else body should contain one or more calls to the same method with argument values that are closer to the stopping condition’s value.” But in the previous section’s line-of-ships example, the flag signals are not likely to contain any values “that are closer to the stopping condition’s value.” In such a case, the stopping condition cannot be computed from method parameters. Identify another way for a recursive method to determine a stopping condition.

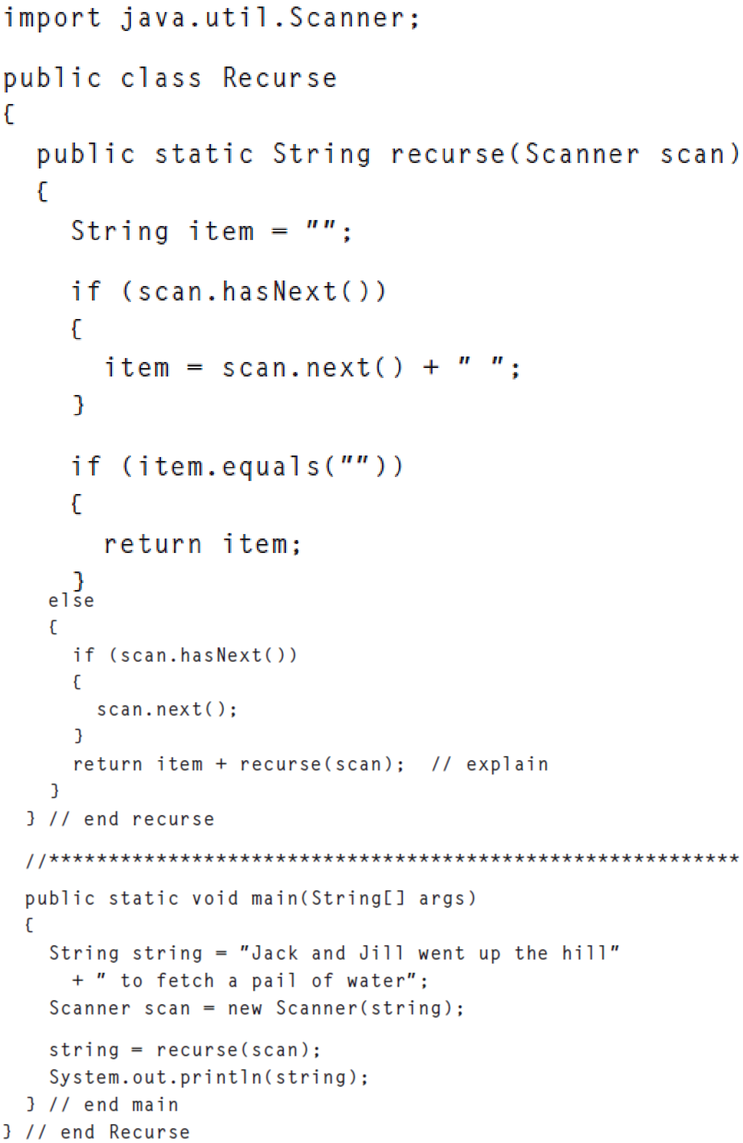
2. [after §11.2] What is wrong with this recursive method?



```
long sum(long n)
{
  return n + sum(n+1);
}
```

3. [after §11.3] Many times, you have used Scanner methods to read and parse keyboard input. You can also use those same

Scanner methods to read and parse an ordinary string. The following program uses the Scanner class's now-familiar next method, but this time it reads from a string instead of the keyboard. When the next method executes, there's an internal cursor that remembers how much of the string has been processed so far and how much has yet to be processed. The program uses the Scanner class's hasNext method to check whether there is any more of the string that has yet to be processed. Study the code. What does the program display for output? Note the recursive return statement that says "// explain." Explain how that statement works.



```
import java.util.Scanner;

public class Recurse
{
  public static String recurse(Scanner scan)
  {
    String item = "";

    if (scan.hasNext())
    {
      item = scan.next() + " ";
    }

    if (item.equals(""))
    {
      return item;
    }
    else
    {
      if (scan.hasNext())
      {
        scan.next();
      }
      return item + recurse(scan);  // explain
    }
  } // end recurse

  //*********************************************************

  public static void main(String[] args)
  {
    String string = "Jack and Jill went up the hill"
      + " to fetch a pail of water";
    Scanner scan = new Scanner(string);

    string = recurse(scan);
    System.out.println(string);
  } // end main
} // end Recurse
```

4. [after §11.4] Modify the previous exercise's Recurse program so that it uses tail recursion. Minimize code changes. You should be able to do this with exactly the same number and sequence of statements and with only internal alterations to one line in the main method and three lines in the recurse method.

5. [after §11.4] Modify the previous exercise's program so that it uses iteration instead of recursion. Do not use an array.

6. [after §11.4] Change Figure 11.2's recursive Factorial program so that it uses tail recursion.

7. [after §11.4] A palindrome is a string that reads the same in both directions. For example, "mom," "kayak," and "avid diva" are all palindromes. Write a recursive method named isPalindrome that determines whether a string is a palindrome. More specifically, your method should return true if the given string parameter (named string) is a palindrome and it should return false otherwise. Use this method signature:

> public static boolean isPalindrome(String string)

Within the recursive method, use String's length method to determine the passed-in string's length, and just before making a tail-recursive call, use one of String's substring methods to remove the parameter string's first and last characters.

8. [after §11.5] What's wrong with this method?

```
public double sum(double x)
{
  if (x >= 0.5)
  {
    return x;
  }
  else
  {
    return sum(x) + Math.random();
  }
} // end sum
```

9. [after §11.5] Given this recurrence relation and stopping condition:

$$y(k, A) = \begin{cases} (1 + A) - A * y(k-1, A) & k > 0 \\ 0 & k <= 0 \end{cases}$$

By hand, evaluate y(3, 0.1), using the format presented in Section 11.5.

10. [after §11.5] Provide a three-parameter tail-recursive static method named fibonacci that generates the numbers in a Fibonacci sequence. It should be an overloaded variation of the one-parameter fibonacci method shown below.

```
import java.util.Scanner;

public class Fibonacci
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);

    System.out.print(
      "Enter max value in Fibonacci sequence: ");
    fibonacci(stdIn.nextInt());
  } // end main

  //*********************************************************

  public static void fibonacci(int max)
  {
    if (max < 0)
    {
      System.out.println("not defined");
    }
    else
    {
      System.out.print("0");
      if (max > 0)
      {
        System.out.print(" 1");
        fibonacci(max, 1, 0);
        System.out.println();
      }
    }
  } // end Fibonacci

  <3-parameter fibonacci method goes here>

} // end class Fibonacci
```

Sample session:

```
Enter max value in Fibonacci sequence: 45
0 1 1 2 3 5 8 13 21 34
```

11. [after §11.5] The following recurrence relation is called the *Logistic Equation*:

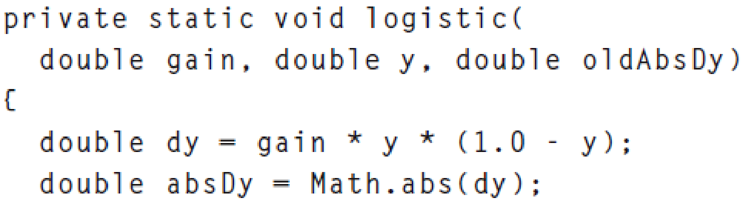
$$y(k) = y(k\text{-}1) + gain * y(k\text{-}1)$$

* (1.0 - y(k-1))

This is the equation used to simulate growth in the optional problem-solving section at the end of Chapter 6. This exercise asks you to use recursion to evaluate successive values generated by the Logistic equation. To get you started, here is a main method that drives recursions for different values of the gain parameter:

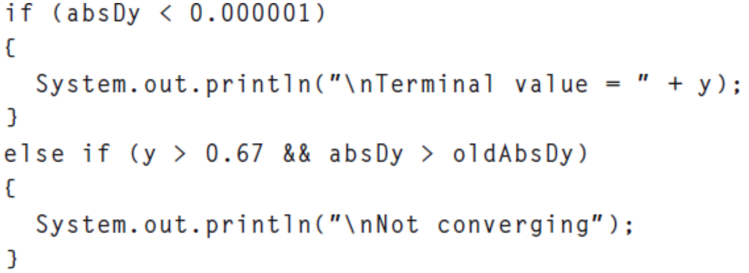
```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);

  // Avoid perfectly periodic solution at gain = 2.0
  for (double gain=0.05; gain<3.5; gain+=0.1)
  {
    System.out.println(”gain = ” + gain);
    logistic(gain, 0.01, 3.0);
    System.out.println();
  } // end for gain
} // end main
```

Provide a Logistic class that includes the above main method plus a static method named logistic that starts like this:



```
private static void logistic(
  double gain, double y, double oldAbsDy)
{
  double dy = gain * y * (1.0 - y);
  double absDy = Math.abs(dy);
```

The logistic method should be tail recursive and include the following terminal conditions:



```
if (absDy < 0.000001)
{
  System.out.println("\nTerminal value = " + y);
}
else if (y > 0.67 && absDy > oldAbsDy)
{
  System.out.println("\nNot converging");
}
```

After getting your program to work, run it with different inputs, and use what you find to describe how y(k) converges for different ranges of gain.

12. [after §11.6] Specify all changes required to make this chapter's binarySearch method search for a particular string in an array of String values.

13. [after §11.7] Modify this chapter's MergeSort class to make it sort the strings in an array of

String values.

14. [after §11.9] Modify the Towers of Hanoi program to help identify its algorithm efficiency. Declare a static variable named moveCount that keeps track of the total number of moves. Remove (or comment out) all the print statements that document a move, and replace them with statements that increment the moveCount variable. In the main method, prompt the user for the number of disks and assign the user input to a local variable named disks. Next, call the move method, but pass disks as an argument instead of the original program's hardcoded 4 argument. After main calls the move method, add a statement that prints the value of disks and the final value of moveCount. Compare those output values with the values in

Table 11.1 and identify the Big O growth rate for the Towers of Hanoi problem.

Review Question Solutions

Page 529

1. A recursive method needs to have at least one parameter whose value changes with successive method calls.

2. False. At a stopping condition, a recursive method returns without making another recursive call.

3. A sufficient stopping condition for the recursive calculation of a factorial is n == 1 || n == 0. An alternate sufficient stopping condition is n <= 1.

4. Before the stopping condition is reached, the value of nF is unknown.

5. If the method's body includes two recursive calls to itself, the method is said to exhibit binary recursion.

6. If it's just as easy, a loop is preferable because it takes less overhead.

7. In the printReverseMessage method, the variable that changes is msg, and the stopping condition is msg.isEmpty();

8. The recurrence relation for a factorial using functional notation is: f(x) = x * f(x-1)

9. The stopping condition is missed if x is less than zero, an odd integer, or not an integer.

10. Because $2^9 = 512$ and $2^{10} = 1024$, it takes 10 recursive method calls to binary-search an array with a length of 1,000.

11. True. Recursive binary search

uses tail recursion, and Figure 9.12 in Section 9.7 shows a straightforward loop implementation.

12. True. The recursive merge sort algorithm uses binary recursion.

13. The stopping condition in a recursive merge sort is when the array length is 1.

14. In a merge sort, when one of the two subordinate arrays becomes empty, the merged array gets all remaining elements from the other subordinate array in their already sorted order.

15. The algorithm implemented by the recursive method that moves a stack of disks is:

- **a)** Move the stack of disks above disk n to the temporary location, t.
- **b)** Move disk n to the

destination location, d.

c) Move the stack of disks moved in (a) from the temporary location to the destination location.

16. In the first recursive call [Review Question Solution 15(a)], use n − 1 for the bottom of the pile to move, use s for the source location, use t for the destination location, and use d for the temporary location.

17. A fractal is an object or quantity that displays *self-similarity* over a range of scales.

18. In our tree-drawing program, model methods specify relative lengths and angles of branches and how trunk size increases in time.

19. In our tree-drawing program, the view specifies screen display at any particular time.

20. In our tree-drawing program, the controller creates and locates new tree instances at particular times and manages the updating of the screen display as time passes.

---

[1] Penultimate means “next to the last.” Isn’t it more fun to say “penultimate” than “next to the last”? Sesquipedalians certainly would agree! ☺

[2] It’s important that we decremented 1039. As we’ll explain later in Section 12.5, if we had tried to use age-- as the final recursive argument, the called method would have used the undecremented age, and recursion would have gone on “forever” until the computer’s stack overflowed.

Type Details and Alternative Coding Mechanisms

Objectives

- Improve your understanding of relationships and differences among primitive data types and your appreciation for their individual limitations.

- Understand how numerical codes identify characters.
- Learn the rules for automatic type conversions and the risks in explicit type casting.
- Understand embedded postfix and prefix increment and decrement operators.
- Understand embedded assignment expressions.
- Learn where and how conditional operator expressions can shorten code.
- See how short-circuit evaluation helps avoid troublesome operations.
- See how empty statement works.
- Learn how to use break statements in loops.
- Create and use an enum.

- Understand Java stream pipelines.
- Program with method references and lambda expressions.
- Optionally, use Unicode characters in GUI applications.

Outline

12.1 Introduction 531

12.2 Integer Types and Floating-Point Types 532

12.3 char Type and the ASCII Character Set 536

12.4 Type Conversions 538

12.5 Prefix/Postfix Modes for Increment/Decrement Operators 541

12.6 Embedded Assignments 544

12.7 Conditional Operator Expressions 546

12.8 Expression Evaluation Review 547

12.9 Short-Circuit Evaluation 551

12.10 Empty Statement 552

12.11 Using break to Exit from a Loop 554

12.12 for Loop Header Details 555

12.13 Enumerated Types 557

12.14 forEach Method, Lambda Expressions, Method References, and Streams 564

12.15 Hexadecimal, Octal, and Binary Numbers 573

12.16 GUI Track: Unicode (Optional) 574

12.17 Introduction to

GridWorld Case Study (Optional) 579

12.1 Introduction

In Chapters 3 and 4, you learned Java language basics. Among other things, you learned about data types, type conversions, and control statements. This chapter describes some additional data types and additional type conversions. It also describes some alternative control statement coding mechanisms.

Chapter 3 introduced you to some of Java's integer and floating-point types of numbers, and Chapter 5 showed you how to find the limits of their ranges. In this chapter, you'll see two more integer types, and for all of the numerical types, you'll learn the amount of storage needed, the precision provided, and how to use range limits. Chapter 3 introduced you to the use of the character type, char. In this chapter, you'll see that each character has an underlying numeric value, and you'll learn how to use those values. Chapter 3 introduced you to type conversion with the cast operator. In this chapter, you'll learn more about type conversions. Chapter 3 introduced you to the increment and decrement operators. In this chapter, you'll discover that you can move the positions of these operators (before or after the variable) to control when they act. Chapter 3 introduced you to assignment operators. In this chapter, you'll see how you can embed assignments within expressions to make code more compact.

Chapter 4 introduced you to several kinds of conditional evaluations. In this chapter, you'll learn about the conditional operator that can take on either of two possible values depending on a boolean condition. You'll also learn about short-circuit evaluation which can prevent errors by stopping a "dangerous" conditional evaluation in certain situations. In addition, you'll learn more about loops. Specifically, you'll see empty-bodied loops and loops that terminate from within the loop's body. And you'll see alternative coding techniques for for loop headers.

The material in this chapter will improve your understanding of several Java nuances and subtleties. This will help you avoid problems in the first place, and it will help you create code that is more efficient and easier to maintain. It will also help you debug code that has problems. It might be your code, or it might be someone else's code. As a real-world programmer, you'll have to work with other people's code, and you'll need to understand what that code is doing.

Much of the material in this chapter could have been inserted at various places earlier in the text. However, it was not necessary for anything we did up until now, and we deferred it until now to keep from encumbering earlier presentations. The assembly of these details into one chapter at this point in the book provides an excellent opportunity for review. As you go through this chapter, integrate this new material into what you learned before and see how it enriches your understanding of those topics.

The last section in this chapter introduces an extended example called GridWorld. The College Board licenses this software to people who want to help new college students obtain advanced placement (AP). It provides a nice example of larger programs, and we will return to it to illustrate new topics as we introduce them.

12.2 Integer Types and Floating-Point Types

This section supplements the numeric data types material you studied in Chapter 3, Section 3.13.

Integer Types

Integer types hold whole numbers (whole numbers are numbers without a decimal point). Figure 12.1 shows the four primitive integer types. The types are ordered in terms of increasing memory storage requirements. Type byte variables require only 8 bits, so they take up the least amount of storage. If you have a program that's taking too much space in memory, you can use smaller types for variables that hold small values. Using smaller types means less storage is needed in memory. Now that memory has become relatively cheap, types byte and short are not used very often.

Type	Storage	Wrapper Class's `MIN_VALUE`	Wrapper Class's `MAX_VALUE`
`byte`	8 bits	−128	127
`short`	16 bits	−32,768	32,767
`int`	32 bits	−2,147,483,648	2,147,483,647
`long`	64 bits	$\approx -9*10^{18}$	$\approx 9*10^{18}$

Figure 12.1 Properties of Java integer data types

To access an integer's minimum and maximum values, use the MIN_VALUE and MAX_VALUE named constants that come with the integer's wrapper class. As you learned in Chapter 5, Integer and Long are the wrapper classes for the int and long data types. And as you might expect, Byte and Short are the wrapper classes for the byte and short data types. So here's how to print the maximum byte value:

```
System.out.println("Largest byte = "
+ Byte.MAX_VALUE);
```

The default type for an integer

constant is int. But you might have a need for an integer constant that is too big for an int. In that case, you can explicitly force an integer constant to be a long by adding an l or L suffix to the integer constant. For example, suppose that you're writing a solar system program, and you want to store the age of the Earth in a variable named ageOfPlanet. The Earth is 4.54 billion years old, and 4.54 billion is larger than Integer.MAX_VALUE's 2,147,483,647. This generates a compilation error:

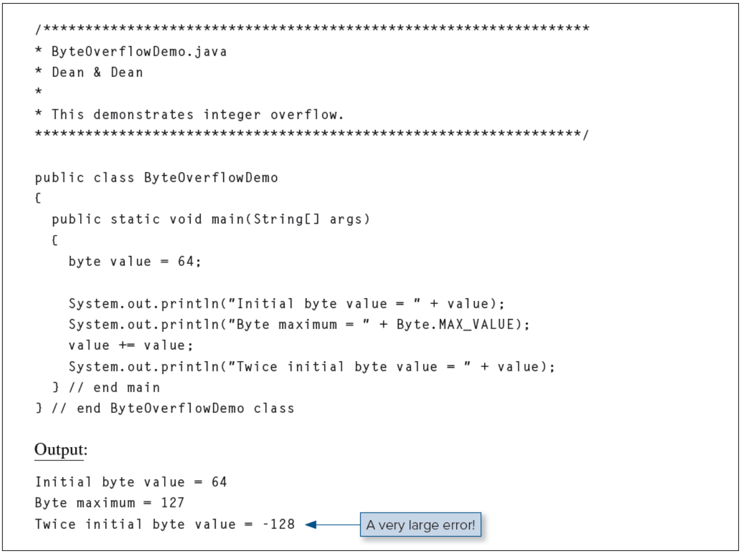
```
long ageOfPlanet = 4_540_000_000;
```

But this, with the L suffix, works just fine:

```
long ageOfPlanet = 4_540_000_000L;
```

⚠ When you declare a numeric variable, be sure that the type you select is large enough to handle the largest value that your program might put into it. If a value can't fit in the memory space provided, that's called *overflow*. Overflow errors are dramatic, as the ByteOverflowDemo program in Figure 12.2 illustrates.

Integer overflow reverses the sign, so the ByteOverflowDemo program prints negative 128 rather than the correct result, positive 128. In this example, the magnitude of the error is approximately twice as big as the magnitude of the largest allowable value! Overflow also causes sign reversal for types short, int, and long. In such cases, the compiler does not find the problem, and the JVM does not find it either. Java runs the program with no complaints and happily generates a massive error. In the end, it's up to you. Whenever there is any doubt, use a larger type!



```
/*********************************************************************
* ByteOverflowDemo.java
* Dean & Dean
*
* This demonstrates integer overflow.
*********************************************************************/

public class ByteOverflowDemo
{
  public static void main(String[] args)
  {
    byte value = 64;

    System.out.println("Initial byte value = " + value);
    System.out.println("Byte maximum = " + Byte.MAX_VALUE);
    value += value;
    System.out.println("Twice initial byte value = " + value);
  } // end main
} // end ByteOverflowDemo class
```

Output:

```
Initial byte value = 64
Byte maximum = 127
Twice initial byte value = -128
```

A very large error!

Figure 12.2 ByteOverflowDemo program illustrates the overflow problem

Floating-Point Types

As you know, floating-point numbers are real numbers—numbers that allow for nonzero digits to the right of a decimal point. This means you can use floating-point numbers to hold fractional values—values that are smaller than one. Figure 12.3 shows the two floating-point types—float and double.

Type	Storage	Precision	Wrapper Class's `MIN_NORMAL`	Wrapper Class's `MAX_VALUE`
`float`	32 bits	6 digits	$\approx 1.2 * 10^{-38}$	$\approx 3.4 * 10^{38}$
`double`	64 bits	15 digits	$\approx 2.2 * 10^{-308}$	$\approx 1.8 * 10^{308}$

Figure 12.3 Properties of Java floating-point data types

Page 534

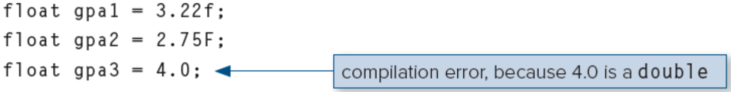
Note Figure 12.3's precision column. Precision refers to the approximate number of digits the type can represent accurately. For example, since float types have 6 digits of precision, if you attempt to store 1.2345678 in a float variable, you would actually store a rounded version—a number like 1.234568. The first 6 digits (1.23456) are precise, but the rest of the number is imprecise. double values have 15 digits of ⚠ precision—quite a bit better than float values with their 6 digits of precision. The relatively low precision of a float can lead to significant round-off errors when you subtract two numbers that are close in value. If the numbers are close enough, then the difference is a very small number where the rightmost digits are merely approximations. This round-off error is compounded when you have repetitive calculations. Because memory is now relatively inexpensive, you should consider float to be an archaic data type, and you should usually avoid it. An exception is when you specify color. Several methods in the Java API Color class employ float type parameters and/or return values.

Be aware that floating-point numbers do worse than integer numbers when it comes to precision. For example, when comparing the 32-bit float type and the 32-bit int type, the floating-point type has less precision. float numbers have 6 digits of precision, whereas int numbers have 9 digits of precision. Likewise, when comparing the 64-bit double type and the 64-bit long type, the floating-point type has less precision. In particular, double numbers have 15 digits of precision, whereas long numbers have 19 digits of precision. Why do floating-point numbers lose out on precision? Some of the bits in floating-point numbers are used to specify the exponent that allows these numbers to take on much greater ranges in magnitude than integer numbers can take on. This reduces the bits available to supply precision.

As you learned in Chapter 5, Float and Double are the wrapper classes for the float and double data types. To access a floating-point data type's minimum and maximum values, use the Float and Double classes' MIN_NORMAL and MAX_VALUE named constants. MAX_VALUE is a floating-point data type's largest positive value, and MIN_NORMAL is a floating-point data type's smallest full-precision positive value. A floating-point's MIN_NORMAL is qualitatively different from an integer's MIN_VALUE. Instead of being a large negative value, a floating-point MIN_NORMAL is a tiny positive fraction. So what are the limits of negative floating-point numbers? The largest-magnitude negative number a floating-point variable can hold is -MAX_VALUE. The smallest-magnitude negative number a floating-point variable can hold safely is -MIN_NORMAL, a tiny negative fraction.

Actually, it's possible for a floating-point variable to hold a number whose magnitude is smaller than MIN_NORMAL. It can hold a value as small as a floating-point MIN_VALUE, which is approximately 1.4 * 10−45 for float and approximately 4.9 * 10−324 for double. But the MIN_VALUE of a floating-point number has only one bit of precision, and that could produce a significant error in a computed result—without any explicit indication that an error is present. This is an example of the worst kind of bug because it can go unrecognized for a long time. Therefore, with floating-point numbers, always use MIN_NORMAL instead of MIN_VALUE.

The default floating-point constant type is double. If you declare a variable to be a float, you must append an f or F suffix to all floating-point constants that go into it, like this:



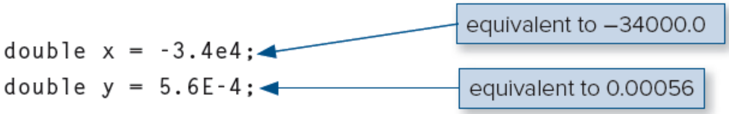
```
float gpa1 = 3.22f;
float gpa2 = 2.75F;
float gpa3 = 4.0;
```

compilation error, because 4.0 is a `double`

⚠ Because of the f and F suffixes, 3.22f and 2.75F are 32-bit float values, so it’s legal to assign them into the 32-bit gpa1 and gpa2 float variables. But 4.0 is a 64-bit double value, and attempting to assign it into the 32-bit gpa3 float variable generates a compilation error.

To write a floating-point number in

scientific notation, put e or E before the base-10 exponent value. If the exponent is negative, insert a minus sign between the e or E and the exponent value. If the exponent is positive, you may use a plus sign after the e or E, but it's not standard practice. In any event, there must never be any whitespace within the number specification. For example:



```
double x = -3.4e4;
double y = 5.6E-4;
```

equivalent to –34000.0

equivalent to 0.00056

BigInteger and BigDecimal

The largest integer type, long, and the largest decimal type, double, are adequate for most applications. However, every now and then, there's an application that needs numbers that are too large for those data types. For example, some of the best data encryption schemes rely on integers that are longer than long. For really large numbers, the java.math package defines the BigInteger and BigDecimal classes. These can represent integer and floating-point numbers of any size.

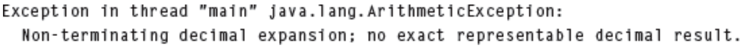
Like an ArrayList, a BigInteger or BigDecimal automatically expands to accommodate its data. BigInteger constructors allow you to create an arbitrarily large integer value with a String, with a byte array, or as a random number with a specified number of bits. The following statement initializes a BigInteger value with a string:

```
BigInteger bigInt = new
BigInteger("24242424242424242424242425
```

BigDecimal constructors allow you to create a decimal value of arbitrary precision with a BigInteger, a long, an int, a double, a String, or a char array. BigInteger and BigDecimal have doubleValue(), intValue(), longValue(), or toString() methods that transform to and return the more familiar date types.

BigInteger mathematical methods include: negate(), add(BigInteger augend), subtract(BigInteger subtrahend), multiply(BigInteger multiplicand), divide(BigInteger divisor), remainder(BigInteger divisor), and pow(int n). BigDecimal provides similar methods, with corresponding BigDecimal arguments.

Unfortunately, some of BigDecimal’s mathematical methods, like divide can generate a result whose fractional digits go on forever, like 0.333333. . . If that happens, the JVM complains:



```
Exception in thread "main" java.lang.ArithmeticException:
  Non-terminating decimal expansion; no exact representable decimal result.
```

To help you avoid this potential problem, BigDecimal provides additional constructors and methods that allow you to specify number of fractional digits (scale), number of significant digits (precision), and rounding technique. For example, here are API headings for two of

BigDecimal's divide methods:

```
public BigDecimal divide(
  BigDecimal divisor, int scale, RoundingMode roundingMode)

public BigDecimal divide(BigDecimal divisor,
  new MathContext(int precision, RoundingMode roundingMode))
```

The scale parameter determines how many digits at the right of the decimal point are in the returned BigDecimal quotient value. The roundingMode parameter is an enumerated type. Enumerated types are discussed later in this chapter, but for now just realize that for standard rounding, use RoundingMode.HALF_UP for the roundingMode parameter's value. The precision parameter determines how many significant digits are in the returned BigDecimal quotient value.

BigInteger, BigDecimal, RoundingMode, and MathContext are all in the java.math package, so that's all you need to import to perform these operations. Although it's cumbersome, BigDecimal gives users extensive control over decimal-point position, total number of significant digits, and rounding behavior—much more control than double. This book contains several examples that use double to represent amounts of money. That's fine for informal work. But double is inappropriate for formal accounting because unpredictable rounding can sometimes produce disturbing output. Typically, accountants like everything to be correct down to the last penny. As such, they would not be pleased with a computer program that displays $1999999999.99 for an amount that is supposed to be exactly $2 billion.

BigDecimal's explicit control over rounding enables you to prevent such aberrations. So, in principle, you could use BigDecimal safely for serious accounting. However, BigDecimal is cumbersome and relatively inefficient. For high-quality financial accounting, you should use integers and represent amounts as cents. To convert floating-point input to an internal integer, multiply the input by 100 and then cast to an integer. To get the result in dollars and cents, use quotient division (i.e., /100) for the dollars part and use modulus division (that is, %100) for the cents part. If int is not big enough, use long. The long type is big enough for even very large financial amounts (e.g., like the total debt of the United States). An ordinary double is good enough for most scientific calculations because it's hard to find measuring instruments that have more precision than double. And science without measurement is not science.

12.3 char Type and the ASCII Character Set

This section supplements the char type material that you studied in

Chapter 3, Section 3.20.

Underlying Numeric Values

For most programming languages, including Java, each character has an underlying numeric value. For example, the character 'A' has the underlying decimal value of 65 and the character 'B' has the underlying value of 66. Most programming languages, including Java, get character numeric values from the *American Standard Code for Information Interchange* (*ASCII*, pronounced "askee") character set. All the characters in the ASCII character set are shown in Appendix 1, in Figures A1.1a and A1.1b.

So what's the point of having underlying numeric values for characters? With underlying numeric values, it makes it easier for the JVM to determine the ordering of characters. For example, because 'A' has the value 65 and 'B' has the value

66, the JVM can easily determine that ‘A’ comes before ‘B’. And knowing the order of characters is necessary for string sort operations. For example, suppose a sort method is given the strings “peach”, “pineapple”, and “apple.” The sort method compares the words’ first characters ‘p’, ‘p’, and ‘a’, and in doing so, the JVM looks up the characters in the ASCII table. Because ‘p’ has the value 112 and ‘a’ has the value 97, “apple” goes first. Then the sort method compares the second characters in “peach” and “pineapple.” Since ‘e’ has the value 101 and ‘i’ has the value 105, “peach” goes before “pineapple.”

You can sometimes use a character’s underlying numeric value to your advantage. For example, suppose a program has a char variable named code whose value determines what happens next. If code holds a ‘p’, the program prints a report, and if code holds a ‘q’, the program quits. Alternatively, if code holds a digit character (‘0’ through ‘9’), the program can perform a mathematical operation, using code’s numeric value as part of that operation. To use code’s numeric value, you first need to retrieve it, which requires a little work. Assuming that digit has been declared as an int variable, here’s how to retrieve code’s numeric value:

```
digit = code - '0';
```

As the ASCII table shows, the zero character (‘0’) has an underlying decimal value of 48. If code holds the six character (‘6’), which has an underlying decimal value of 54, then 6 will be assigned to the digit variable, and all is well. This character-to-number code fragment works because the ASCII character set uses successive underlying numeric values (48 through

57) for the digit characters ‘0’ through ‘9’.

As an alternative to the code fragment above, you could accomplish the same thing with this:

```
digit = code - 48;
```

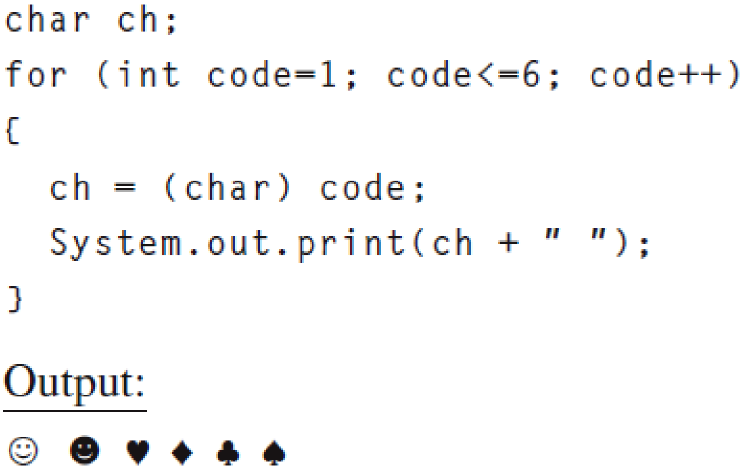
However, using 48 in your program would be considered inelegant. Why? Because it leads to code that is hard to understand (for someone to understand it, he or she would have to know that 48 is the underlying value of the ‘0’ character). Using ‘0’, on the other hand, would be easier to understand. It’s an example of self-documenting code.

Control Characters

Most characters in the ASCII character set represent printable symbols. For example, the ‘f’ character represents the printable letter *f*. But the first 32 characters and the last character in the

ASCII character set are different—they are *control characters*. Control characters perform nonprinting operations. For example, the start-of-heading character (ASCII numeric value 1) helps with data being sent from one computer device to another. More specifically, it signals the beginning of transmitted data. When you print a control character, you might be surprised by what appears on the screen. The bell character (ASCII numeric value 7) normally generates a sound and displays nothing, which makes sense, but the start-of-heading character displays something less intuitive. When you print the start-of-heading character, you'll get different results in different environments. For example, in a console window[1] in a Windows environment, a smiley face is displayed. In other environments, a blank square is displayed. Note the following code fragment, with -

associated output from a console window in a Windows environment:



```
char ch;
for (int code=1; code<=6; code++)
{
  ch = (char) code;
  System.out.print(ch + " ");
}
```

Output:

```
☺ ☻ ♥ ♦ ♣ ♠
```

In the above code fragment, the (char) cast operator uses the ASCII table to return the character associated with code's numeric value. So if code has the value 1, then (char) code returns the start-of-heading character.

Unicode Character Set

The ASCII character set works just fine most of the time, but in some situations, it's insufficient. Sometimes you'll need characters and symbols that are outside the ASCII character set. For example, suppose you want to display a check mark (✓) or the pi symbol (π). Those two characters are not part of ASCII. They are part of a newer coding scheme called *Unicode,* which is a superset of ASCII. You can learn about Unicode in the optional section at the end of this chapter (Section 12.16). In that section, we show you how to access the check mark and pi symbols and the many other characters enumerated in the Unicode Standard.

Using the + Operator with chars

Remember how you can use the + operator to concatenate two strings together? You can also use the + operator to concatenate a char to a string. Note this example:

```
char first = 'J';
char last = 'D';
System.out.println("Hello, " + first + last + '!');
```

Output:

```
Hello, JD!
```

When the JVM sees a string next to a + sign, it concatenates by first converting the operand on the other side of the + sign to a string. So in the above example, the JVM converts the first variable to a string and then concatenates the resulting “J” to the end of “Hello, ” to form “Hello, J”. The JVM does the same thing with each of the next two characters it sees, last’s stored character and ‘!’. It converts each one to a string and concatenates each one to the string at its left.

Be aware that if you apply the + operator to two characters, the + operator does not perform concatenation; instead, it performs mathematical addition using the characters’ underlying ASCII values. Note this example:

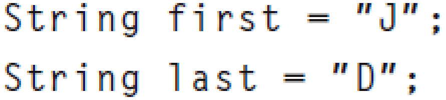
```
char first = 'J';
char last = 'D';
System.out.println(first + last + ", What's up?");
```

Output:

```
142, What's up?
```

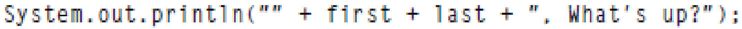
The intended output is: JD, What's up? Why does the code fragment print 142 instead of JD? The JVM evaluates + operators (and most other operators as well) left to right, so in evaluating println’s argument, it first evaluates first + last. Because both first and last are char variables, the JVM performs mathematical addition using the characters’ underlying ASCII values. first holds ‘J’ and J’s value is 74. last holds ‘D’ and D’s value is 68. So first + last evaluates to 142.

There are two ways to fix the above code. You can change the first two lines to string initializations like this:



```
String first = "J";
String last = "D";
```

Or you can insert an empty string at the left of println’s argument like this:



```
System.out.println("" + first + last + ", What's up?");
```

12.4 Type Conversions

This section supplements the type casting material you studied in Section 3.19.

Java is a *strongly typed* language, so each variable and each value within a program is defined to have a particular data type. As with all strongly typed languages, you need to be careful when working with more than one data type. In this section, you learn how some, but not all, data types convert to other data types. Java makes some type conversions automatically, and it allows you to force some other type conversions. Either way, be careful. Inappropriate type conversions can cause problems.

To figure out what's allowed in terms of type conversions, learn the ordering scheme in Figure 12.4. Crudely speaking, this picture shows what types can "fit inside" other types. For example, a byte value with 8 bits can fit inside a short variable that holds 16 bits because an 8-bit entity is “narrower” than a 16-bit entity. We like the terms narrower and wider to describe type sizes, but be aware that those are not formal terms; other people do not use those terms. Notice that the boolean type does not appear in this picture. You cannot convert between numeric types and the boolean type.

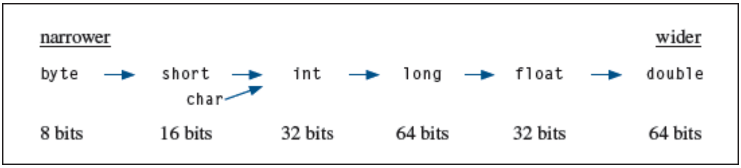

narrower
wider
byte
short
char
int
long
float
double
8 bits
16 bits
32 bits
64 bits
32 bits
64 bits

Figure 12.4 Type conversion ordering scheme

Promotion

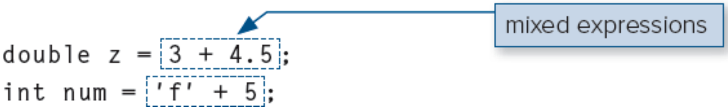
There are two kinds of type conversion —*promotion* (automatic type conversion) and *type casting* (forced type conversion). You’ve already seen type casting. We’ll revisit it shortly, but let’s first discuss promotion.

A promotion is an implicit conversion. It's when an operand's type is automatically converted without having to use a cast operator. It occurs when there's an attempt to use a narrower type in a place that expects a wider type; that is, it occurs when you're going with the flow of the arrows in Figure 12.4. Promotion often occurs in assignment statements. If the expression on the right of an assignment statement evaluates to a type that is narrower than the type of the variable on the left of the assignment statement, then during the assignment the narrower type on the right gets promoted to the wider type on the left. Note these promotion examples:

```
long x = 44;
float y = x;
```

In the first statement, 44 is an int. The int 44 is narrower than the long x, so the JVM promotes 44 to a long, and then performs the assignment. In the second assignment statement, x is a long. The long x is narrower than the float y, so the JVM promotes x to a float, and then performs the assignment.

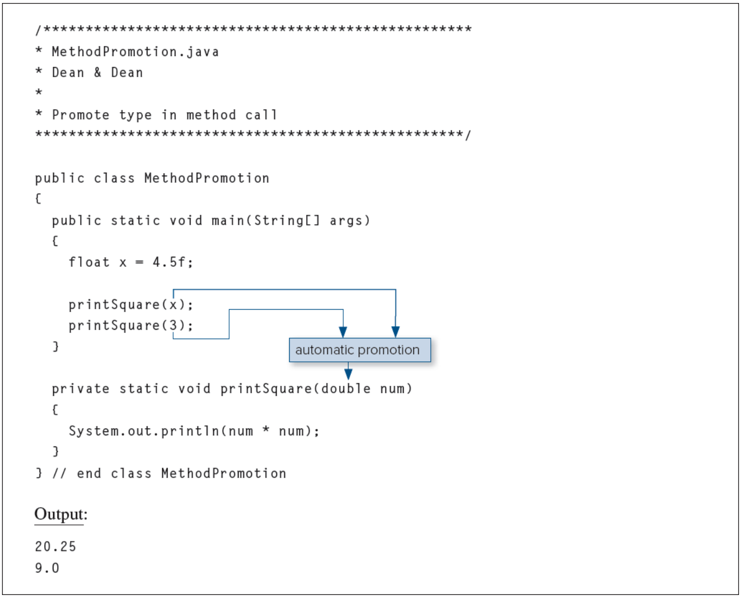
Note these additional promotion examples:



```
double z = 3 + 4.5;
int num = 'f' + 5;
```

The expressions on the right are *mixed expressions*. A mixed expression is an expression that contains operands of different data types. Within a mixed expression, the narrower operand automatically promotes to the type of the wider operand. In the first statement above, the int 3 is narrower than the double 4.5, so the JVM promotes 3 to a double before adding it to 4.5. In the second statement above, do you know which operand, ‘f’ or 5, gets promoted to match the other one? ‘f’ is a char and 5 is an int, and Figure 12.4 shows that char is narrower than int. Thus, the JVM promotes ‘f’ to an int. More specifically, since f’s underlying numeric value is 102 (see Figure A1.1b), the JVM promotes ‘f’ to 102. Then the JVM adds 102 to 5 and assigns the resulting 107 to num.

Promotions typically occur as part of assignment statements, mixed expressions, and method calls. You’ve already seen examples with assignment statements and mixed expressions; now let’s examine promotions with method calls. As mentioned above, conversions take place any time there’s an attempt to use a narrower type in a place that expects a wider type. So if you pass an argument to a method and the method’s parameter is defined to be a wider type than the argument’s type, the argument’s type promotes to match the parameter’s type. Figure 12.5’s program provides an example of this behavior. Can you determine what promotion takes place within the program? The x argument is a float and it promotes to a double. The 3 argument is an int and it promotes to a double as well.



```
/****************************************************
* MethodPromotion.java
* Dean & Dean
*
* Promote type in method call
****************************************************/

public class MethodPromotion
{
  public static void main(String[] args)
  {
    float x = 4.5f;

    printSquare(x);
    printSquare(3);
  }

  private static void printSquare(double num)
  {
    System.out.println(num * num);
  }
} // end class MethodPromotion
```

automatic promotion

Output:

```
20.25
9.0
```

Figure 12.5 Program that demonstrates type promotion in a method call

Type Casting

Type casting is an explicit type

conversion. It occurs when you use a cast operator to convert an expression's type. Here's the syntax for using a cast operator:

(*type*) *expression*

It's legal to use a cast operator to convert any numeric type to any other numeric type; that is, the conversion can go in either direction in Figure 12.4's ordering-scheme diagram. For example, the following code fragment casts the double x to the int y:

```
double x = 12345.6;
int y = (int) x;
System.out.println("x = " + x + "\ny = " + y);
```

What happens if you omit the (int) cast operator? You'd get a compilation error because you'd be directly assigning a double into an int and that's forbidden (in Figure 12.4's ordering-scheme diagram, there's no arrow going from the double type to the int type). Why is it illegal to assign a floating-point number directly into an int? Because floating-point numbers can have fractions, and ints can't handle fractions.

Do you know what the above code fragment prints? x remains unchanged (even though (int) was applied to it), and y gets the whole-number portion of x with x's fraction truncated, not rounded. So here's the output:

```
x = 12345.6
y = 12345
```

The program in Figure 12.6 further illustrates the use of cast operators. It prompts the user to enter an ASCII value (an integer between 0 and 127). Then it prints the character associated with that ASCII value and also the next character in the ASCII table. In the program, what do the two cast operators do? The first one returns the char version of asciiValue, an int variable. The second one returns the char version of asciiValue + 1. The cast operations are needed to print ch and nextCh as characters, rather than integers. ⚠ What would happen if you omitted the cast operators? You'd get compile-time errors because you'd be assigning an int variable directly into a char, and that's forbidden according to the ordering scheme in Figure 12.4.

Why is it illegal to assign the contents of a number variable directly into a char variable? You'd think it would be safe to assign a small whole number, like a byte with 8 bits, into a char with 16 bits. It's illegal to assign the contents of a number variable directly into a char because numbers can be negative and a char can't handle negativity (a char variable's underlying value is a positive number between 0 and 65535).

12.5 Prefix/Postfix Modes for Increment/Decrement Operators

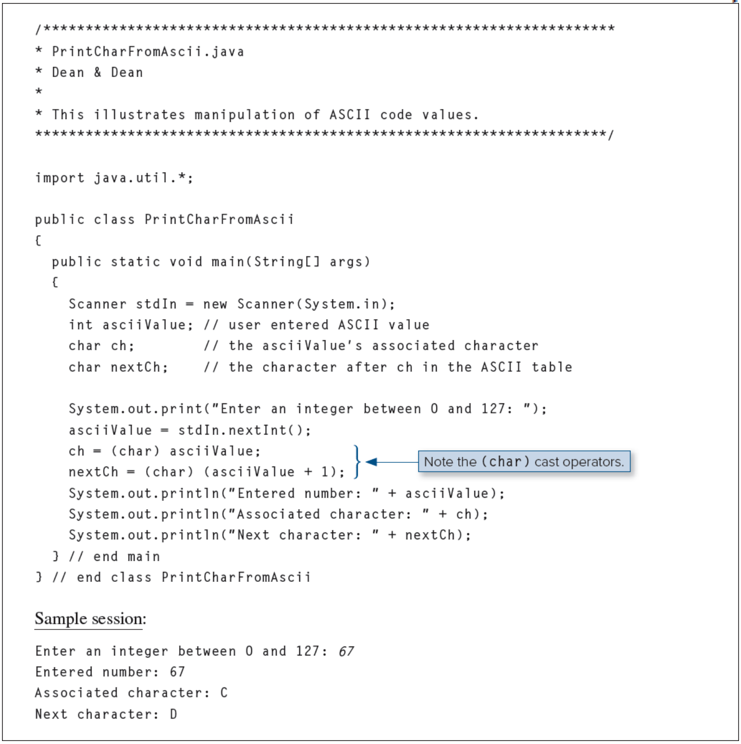
This section supplements material you studied in the first part of Chapter 3, Section 3.17 (Increment and Decrement Operators), and it uses techniques you studied in Chapter 3, Section 3.18 (Tracing).

The increment operator has two different modes—the *prefix mode* and the *postfix mode*. The prefix mode is when you put the ++ before the variable that is to be incremented. Using the prefix mode causes the variable to be incremented before the variable's value is used. For example:

```
y = ++x       is equivalent to      x = x + 1;
                                    y = x;
```

The postfix mode is when you put the ++ after the variable that is to be incremented. Using the postfix mode causes the variable to be incremented after the variable's value is used. For example:

`y = x++` is equivalent to `y = x;`
`x = x + 1;`



```
/********************************************************************
* PrintCharFromAscii.java
* Dean & Dean
*
* This illustrates manipulation of ASCII code values.
********************************************************************/

import java.util.*;

public class PrintCharFromAscii
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int asciiValue; // user entered ASCII value
    char ch;        // the asciiValue's associated character
    char nextCh;    // the character after ch in the ASCII table

    System.out.print("Enter an integer between 0 and 127: ");
    asciiValue = stdIn.nextInt();
    ch = (char) asciiValue;
    nextCh = (char) (asciiValue + 1);
    System.out.println("Entered number: " + asciiValue);
    System.out.println("Associated character: " + ch);
    System.out.println("Next character: " + nextCh);
  } // end main
} // end class PrintCharFromAscii
```

Note the `(char)` cast operators.

Sample session:

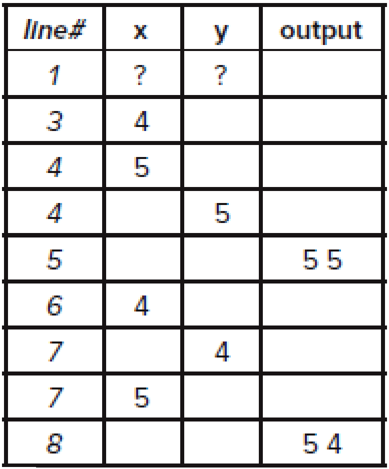
```
Enter an integer between 0 and 127: 67
Entered number: 67
Associated character: C
Next character: D
```

Figure 12.6 Program illustrating the use of cast to convert character codes into characters

To get a better feeling for how this works, trace this code fragment:

```
1   int x, y;
2
3   x = 4;
4   y = ++x;
5   System.out.println(x + " " + y);
6   x = 4;
7   y = x++;
8   System.out.println(x + " " + y);
```

Here is the trace:



line#	x	y	output
1	?	?	
3	4		
4	5		
4		5	
5			5 5
6	4		
7		4	
7	5		
8			5 4

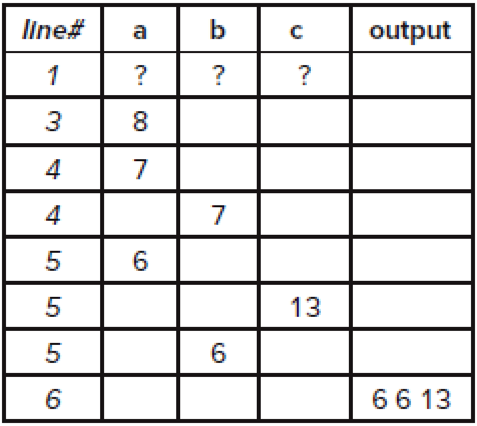
Pay attention to the quotes.

Here's a review question to help with your debugging skills. What would the outputs have been if the println arguments had been (x + ' ' +

y)? Instead of specifying the string version of a space, this would have specified the character version of a space, and it would make the computer consider the argument to be a mathematical expression rather than a string concatenation. Because x and y are integers, it would promote the space character to its underlying numeric value, which is 32 (see Figure A1.1b in Appendix 1). The first print statement would add (5 + 32 + 5) and print 42. The second statement would add (5 + 32 + 4) and print 41.

The decrement operator's prefix and postfix modes work the same as for the increment operator, but they subtract 1 instead of adding 1. To get a feeling for how they work, trace this code fragment:

```
int a, b, c;

a = 8;
b = --a;
c = b-- + --a;
System.out.println(a + " " + b + " " + c);
```



line#	a	b	c	output
1	?	?	?	
3	8			
4	7			
4		7		
5	6			
5			13	
5		6		
6				6 6 13

Let’s examine line 5 in more depth:

c = b-- + --a;

As you might have guessed, in executing this statement, the JVM first decrements a. This should make sense when you look at the operator precedence table in Appendix 1 and confirm that the decrement operator has very high precedence. The JVM also executes b’s decrement operator early on, but its execution consists of using b’s original value and incrementing b afterwards. The operator precedence table shows that the + operator has higher precedence than the = operator, so the JVM next adds b's original value to a's decremented value. Finally, the JVM assigns the sum to c.

For many people, line 5 is particularly confusing. We showed you this example because you might see this kind of thing in someone else's code, but if you want your code to be understandable, we recommend that you not do this yourself. That is, don't embed ++ or -- expressions within other expressions. Instead of trying to do everything line 5 does in one statement, it would be more understandable to partition line 5 into three separate statements, like this:

```
5a a--;
5b c = b + a;
5c b--;
```

The JVM performs the evaluation in separate steps anyway, so writing it out does not incur any performance penalty. It takes more space on the page, but most people will agree that it's easier to read.

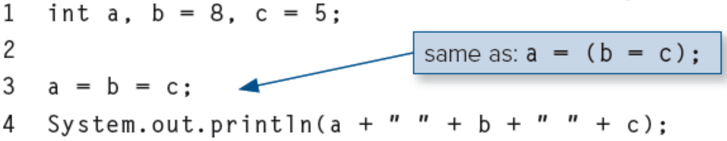
When writing code, how do you decide which mode to use, prefix or postfix? It depends on the rest of your code. Usually, to minimize confusion, you'll put increment and decrement operations on separate lines. Then it doesn't matter which mode you use, but postfix is more common.

12.6 Embedded Assignments

This section supplements material you learned in Chapters 3 and 4. Specifically, it supplements the -assignment statements material in Section 3.11 and the while loop material in Section 4.8.

Embedding an Assignment within Another Assignment

Assignments are sometimes embedded as expressions in larger statements. When that happens, remember that (1) an assignment expression evaluates to the assigned value, and (2) assignment operators exhibit right-to-left associativity. To see these concepts in action, consider this code fragment:



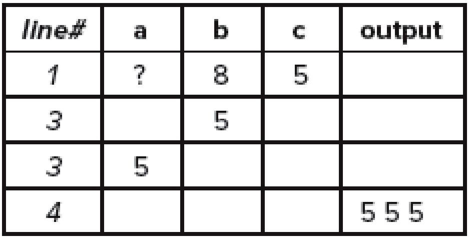
```
1  int a, b = 8, c = 5;
2
3  a = b = c;
4  System.out.println(a + " " + b + " " + c);
```

same as: `a = (b = c);`

Line 3 shows an assignment expression embedded inside a larger assignment statement. Which of the two assignment operators does the JVM execute first? Because assignment operators exhibit right-to-left associativity, the JVM executes the right assignment operation first. What does the b = c expression evaluate to? It evaluates to 5 because the assigned value, c, is 5. In evaluating line 3, replace the b = c part of the statement with 5 to reduce the statement to:

```
a = 5;
```

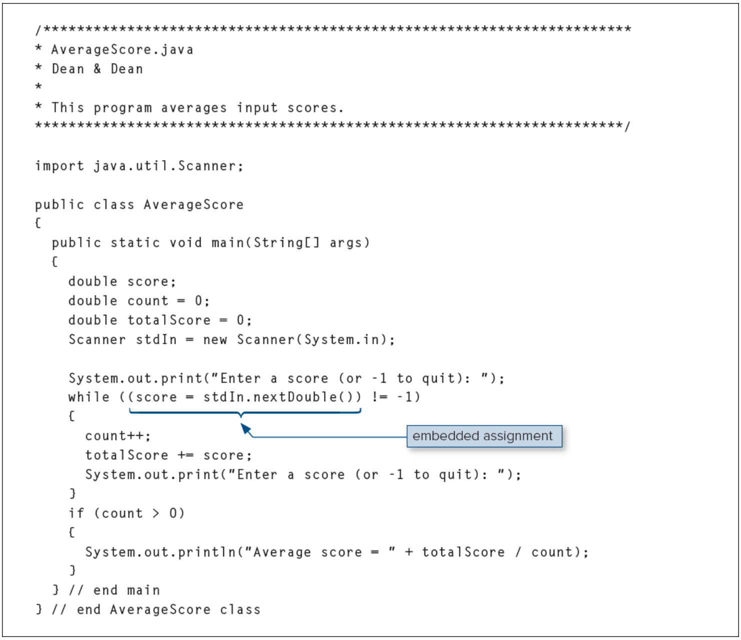
Here's what the code fragment's trace looks like:



line#	a	b	c	output
1	?	8	5	
3		5		
3	5			
4				5 5 5

Embedding an Assignment Within a Loop Condition

Except for a pure multiple assignment like a = b = c; it's best to avoid embedding multiple assignments as expressions in other statements, because that makes code hard to understand. Nevertheless, it's fairly common to embed a single assignment as an expression in a loop condition. For example, Figure 12.7 - contains a program that averages a set of input scores. Note the (score = stdIn.nextDouble()) assignment inside the while condition. If, for example, the user responds to the prompt by entering 80, score gets the value 80, the assignment expression within the parentheses evaluates to 80, and the while loop header becomes:



```
/***********************************************************************
* AverageScore.java
* Dean & Dean
*
* This program averages input scores.
***********************************************************************/

import java.util.Scanner;

public class AverageScore
{
  public static void main(String[] args)
  {
    double score;
    double count = 0;
    double totalScore = 0;
    Scanner stdIn = new Scanner(System.in);

    System.out.print("Enter a score (or -1 to quit): ");
    while ((score = stdIn.nextDouble()) != -1)
    {
      count++;
      totalScore += score;
      System.out.print("Enter a score (or -1 to quit): ");
    }
    if (count > 0)
    {
      System.out.println("Average score = " + totalScore / count);
    }
  } // end main
} // end AverageScore class
```

embedded assignment

Figure 12.7 AverageScore program that demonstrates use of embedded assignments

while (80 != -1)

Because the condition is true, the JVM executes the body of the loop. If the assignment expression were not embedded in the while loop condition, it would have to appear twice—once above the loop header and again at the bottom of the loop. Embedding the assignment in the condition improves the loop's structure.

You will sometimes also see embedded assignments in method arguments and array indices. This makes code more compact. Compactness is often a good thing in that it can lead to code that is less cluttered and therefore easier to understand. But don't go too far in trying to make your code compact because compactness can sometimes lead to code that is harder to understand (i.e., it can lead to code that is more *cryptic*). Some programmers get a kick out of making "clever" programs that are as compact as possible. If that's you, try to redirect your efforts to making programs as understandable as possible. You can still use compact code, but do so in a manner that helps, not hinders, understandability.

12.7 Conditional Operator Expressions

This section supplements the material in Chapter 4, Section 4.3 (if Statements).

Syntax and Semantics

When you want a logical condition to determine which of two alternate values applies, instead of using the "if, else" form of the if statement, you can use a conditional operator expression. The conditional operator is Java's only *ternary* operator. Ternary means three. The conditional relates three operands with the two symbols, ? and :. The ? goes between the first and second operands, and the : goes between the second and third operands.

Here's the syntax:

condition ? *expression1* : *expression2*

If the condition is true, the conditional operator expression evaluates to the value of *expression1,* and it ignores *expression2*. If the condition is false, the conditional operator expression evaluates to the value of *expression2,* and it ignores *expression1*. Think of *expression1* as the true part of an "if, else" statement. Think of *expression2* as the false part of an "if, else" statement.

For example, consider this expression:

(x>y) ? x : y

The parentheses around the condition are not required, because > has higher precedence than the ?: pair, but we recommend using them because they improve readability. What does the JVM do when it sees this expression?

- It compares x with y.
- If x is greater, it evaluates the expression to x.
- If x is not greater, it evaluates the expression to y.

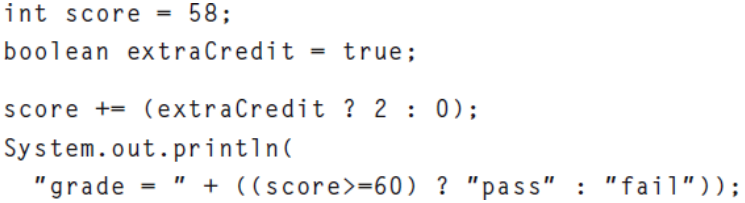
Do you know what general functionality the expression implements? It finds the maximum between two numbers. You can prove this to yourself by plugging in sample numbers. Suppose x equals 2 and y equals 5. Here's how the expression evaluates to the maximum, 5:

$(2 > 5) ? 2 : 5 \Rightarrow$
$(\text{false}) ? 2 : 5 \Rightarrow$
5

Using the Conditional Operator

A conditional operator expression

cannot appear on a line by itself because it is not a complete statement. It is just part of a statement—an expression. The following code fragment includes two examples of embedded conditional operator expressions:



```
int score = 58;
boolean extraCredit = true;

score += (extraCredit ? 2 : 0);
System.out.println(
  "grade = " + ((score>=60) ? "pass" : "fail"));
```

How does it work? Because extraCredit is true, the first conditional operator evaluates to 2. score then increments by 2 from its initial value of 58 to 60. Because (score> =60) evaluates to true, the second conditional operator evaluates to “pass.” The println statement then prints:

```
grade = pass
```

In the above code fragment, we like the parentheses the way they are shown, but in the interest of honing your debugging skills, let’s examine what happens if you omit each of the pairs of parentheses. As shown in the operator precedence table in Appendix 2, the conditional operator has higher precedence than the += operator. Therefore, it would be legal to omit the parentheses in the += assignment statement. In the println statement, the conditional operator has lower precedence than the + operator, so you must keep the parentheses that surround the conditional operator expression. Because the >= operator has higher precedence than the conditional operator, it would be legal to omit the parentheses that surround the score>=60 condition. Note how we omit spaces in the score>=60 condition but include spaces around the ? and : that separate the three components of the conditional operator expression. This style improves readability.

You can use the conditional operator

to avoid if statements. Conditional operator code might look more efficient than if statement code because the source code is shorter, but the generated bytecode is typically longer. This is another example of something you might see in someone else's code, but because it's relatively hard to understand, we recommend that you use it with restraint in your own code. For example, the score += (extraCredit ? 2 : 0); statement in the above code fragment is rather cryptic. It would be better style to increment the score variable like this:

```
if (extraCredit)
{
  score += 2;
}
```

12.8 Expression Evaluation Review

Hand calculation helps you understand.

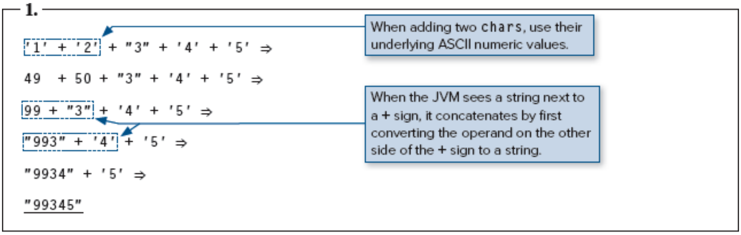
So far in this chapter, you've learned quite a few type details and operator details. Learning such details will help you debug code that has problems, and it will help you avoid problems in the first place. To make sure that you really understand the details, let's do some expression evaluation practice problems.

Expression Evaluation Practice with Characters and String Concatenation

Note the following three expressions. Try to evaluate them on your own prior to looking at the subsequent answers. While performing the evaluations, remember that if you have two or more operators with the same precedence, use left-to-right associativity (i.e., perform the operation at the left first). So in the first expression, you should perform the + operation in '1' + '2' before attempting to perform the second + operation.

1. '1' + '2' + "3" + '4' + '5'
2. 1 + 2 + "3" + 4 + 5
3. 1 + '2'

Here are the answers:

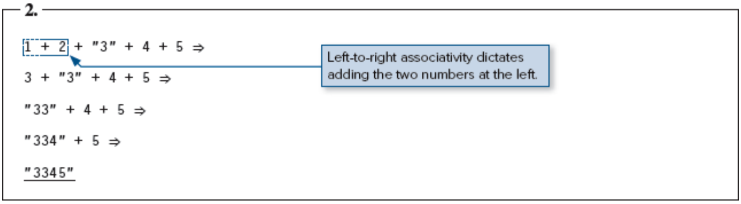


1.

```
'1' + '2' + "3" + '4' + '5' ⇒
49  + 50 + "3" + '4' + '5' ⇒
99 + "3" + '4' + '5' ⇒
"993" + '4' + '5' ⇒
"9934" + '5' ⇒
"99345"
```

When adding two chars, use their underlying ASCII numeric values.

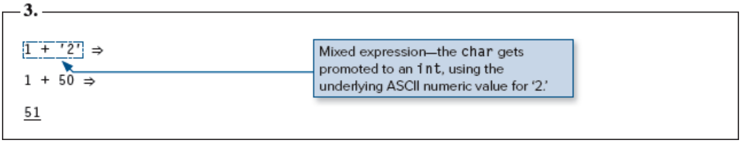
When the JVM sees a string next to a + sign, it concatenates by first converting the operand on the other side of the + sign to a string.



2.

```
1 + 2 + "3" + 4 + 5 ⇒
3 + "3" + 4 + 5 ⇒
"33" + 4 + 5 ⇒
"334" + 5 ⇒
"3345"
```

Left-to-right associativity dictates adding the two numbers at the left.



3.

```
1 + '2' ⇒
1 + 50 ⇒
51
```

Mixed expression—the char gets promoted to an int, using the underlying ASCII numeric value for '2.'

Expression Evaluation Practice with Type Conversions and

Various Operators

Assume the following:

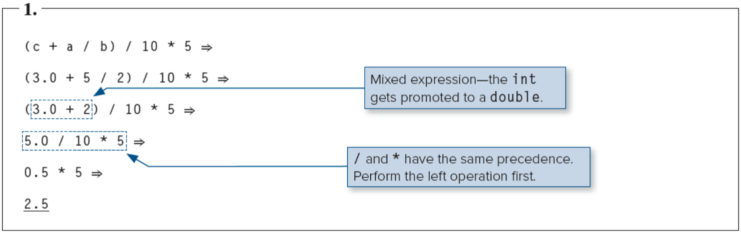
```
int a = 5, b = 2;
double c = 3.0;
```

Try to evaluate the following expressions on your own prior to looking at the subsequent answers:

1. (c + a / b) / 10 * 5
2. a + b++
3. 4 + --c
4. c = b = a % 2

Here are the answers:

1.

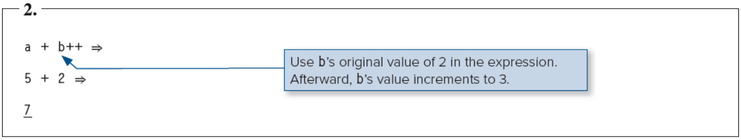


```
(c + a / b) / 10 * 5 ⇒
(3.0 + 5 / 2) / 10 * 5 ⇒
(3.0 + 2) / 10 * 5 ⇒
5.0 / 10 * 5 ⇒
0.5 * 5 ⇒
2.5
```

Mixed expression—the `int` gets promoted to a `double`.

/ and * have the same precedence. Perform the left operation first.

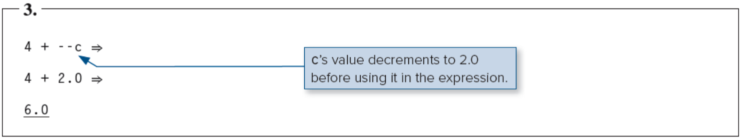
2.



```
a + b++ ⇒
5 + 2 ⇒
7
```

Use b's original value of 2 in the expression. Afterward, b's value increments to 3.

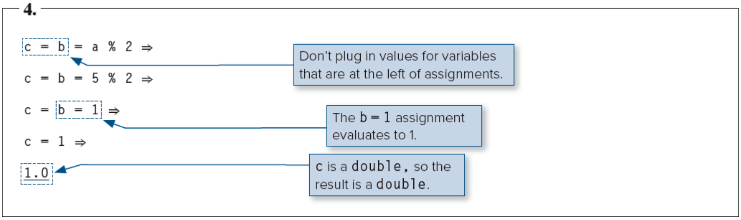
3.



```
4 + --c ⇒
4 + 2.0 ⇒
6.0
```

c's value decrements to 2.0 before using it in the expression.

4.



```
c = b = a % 2 ⇒
c = b = 5 % 2 ⇒
c = b = 1 ⇒
c = 1 ⇒
1.0
```

Don't plug in values for variables that are at the left of assignments.

The b = 1 assignment evaluates to 1.

c is a `double`, so the result is a `double`.

More Expression Evaluation Practice

Assume the following:

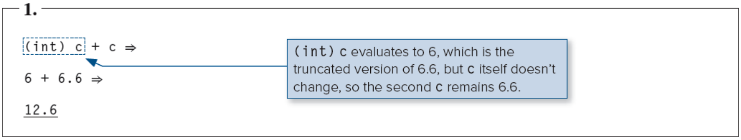
```
int a = 5, b = 2;
double c = 6.6;
```

Try to evaluate the following expressions on your own prior to looking at the subsequent answers:

1. (int) c + c
2. b = 2.7
3. ('a' < 'B') && ('a' == 97) ? "yes" : "no"
4. (a > 2) && (c = 6.6)

Here are the answers:

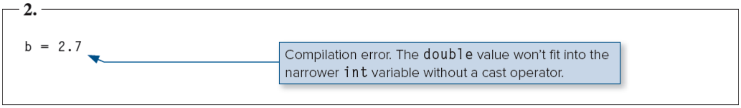
1.



```
(int) c + c ⇒
6 + 6.6 ⇒
12.6
```

(int) c evaluates to 6, which is the truncated version of 6.6, but c itself doesn't change, so the second c remains 6.6.

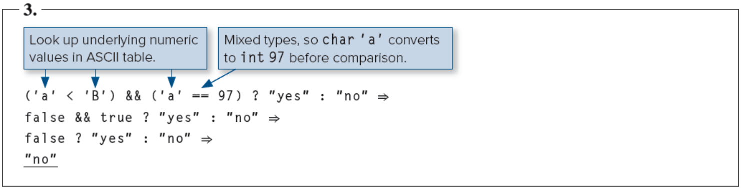
2.



```
b = 2.7
```

Compilation error. The double value won't fit into the narrower int variable without a cast operator.

3.

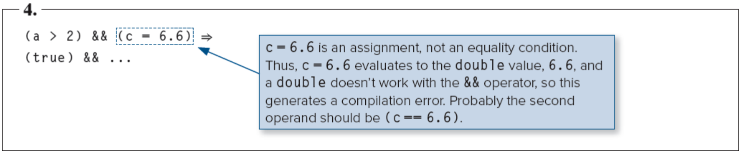


Look up underlying numeric values in ASCII table.

Mixed types, so char 'a' converts to int 97 before comparison.

```
('a' < 'B') && ('a' == 97) ? "yes" : "no" ⇒
false && true ? "yes" : "no" ⇒
false ? "yes" : "no" ⇒
"no"
```

4.



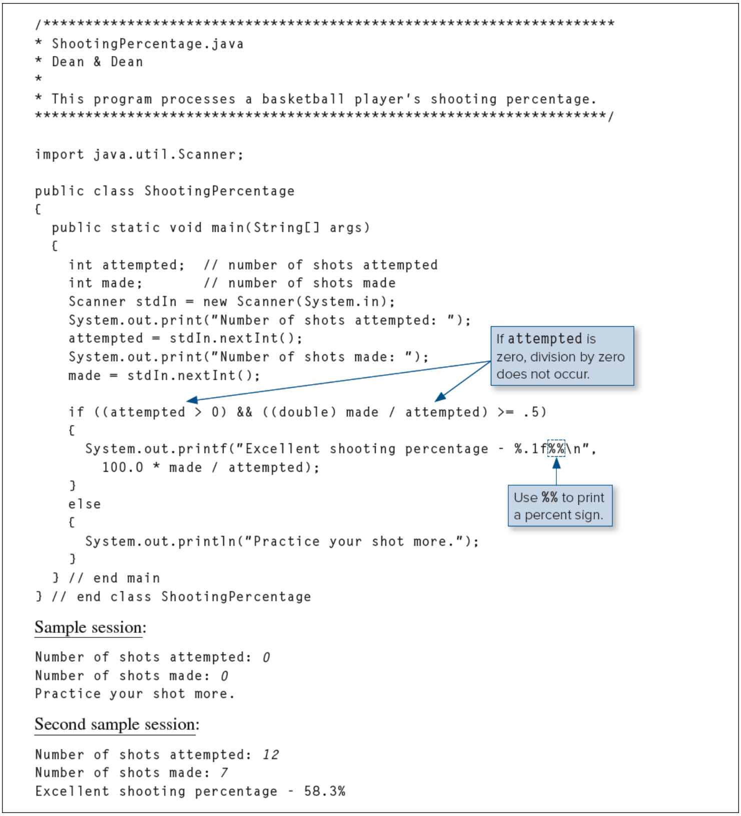
```
(a > 2) && (c = 6.6) ⇒
(true) && ...
```

c = 6.6 is an assignment, not an equality condition. Thus, c = 6.6 evaluates to the double value, 6.6, and a double doesn't work with the && operator, so this generates a compilation error. Probably the second operand should be (c == 6.6).

12.9 Short-Circuit Evaluation

This section supplements the && logical operator material you studied in Chapter 4, Section 4.4, and the || logical operator material you studied in Chapter 4, Section 4.5.

Consider the program in Figure 12.8. It calculates a basketball player's shooting percentage and prints an associated message. Note the if statement's heading, repeated here for your convenience. In particular, note the division operation with attempted in the denominator.



```
/*********************************************************************
* ShootingPercentage.java
* Dean & Dean
*
* This program processes a basketball player's shooting percentage.
*********************************************************************/

import java.util.Scanner;

public class ShootingPercentage
{
  public static void main(String[] args)
  {
    int attempted;  // number of shots attempted
    int made;       // number of shots made
    Scanner stdIn = new Scanner(System.in);
    System.out.print("Number of shots attempted: ");
    attempted = stdIn.nextInt();
    System.out.print("Number of shots made: ");
    made = stdIn.nextInt();

    if ((attempted > 0) && ((double) made / attempted) >= .5)
    {
      System.out.printf("Excellent shooting percentage - %.1f%%\n",
        100.0 * made / attempted);
    }
    else
    {
      System.out.println("Practice your shot more.");
    }
  } // end main
} // end class ShootingPercentage
```

If `attempted` is zero, division by zero does not occur.

Use `%%` to print a percent sign.

Sample session:

```
Number of shots attempted: 0
Number of shots made: 0
Practice your shot more.
```

Second sample session:

```
Number of shots attempted: 12
Number of shots made: 7
Excellent shooting percentage - 58.3%
```

Figure 12.8 Program that illustrates short-circuit evaluation

if ((attempted > 0) && ((double) made / attempted) > = .5)

With division, you should always think about, and try to avoid, division by zero. If attempted equals zero, will the JVM attempt to divide by zero? Nope!

Short-circuit evaluation saves the day.

Short-circuit evaluation means that the JVM stops evaluating an expression whenever the expression's outcome becomes certain. More specifically, if the left side of an && expression evaluates to false, then the expression's outcome is certain (false && anything evaluates to false) and the right side is skipped. Likewise, if the left side of an || expression evaluates to true, then the expression's outcome is certain (true || anything evaluates to true) and the right side is skipped. So in Figure 12.8's if statement condition, if attempted equals zero, the left side of the && operator evaluates to false and the right side is skipped, thus avoiding division by zero.

So what's the benefit of short-circuit evaluation?

Utilize built-in behavior.

1. Error avoidance: It can help to prevent problems by enabling you

to avoid an illegal operation on the right side of an expression.

2. Performance: Because the result is already known, the computer doesn't have to waste time calculating the rest of the expression.

As an aside, note the %% in Figure 12.8's printf statement. It's a conversion specifier for the printf method. Unlike the other conversion specifiers, it is a stand-alone entity; it doesn't have an argument that plugs into it. It simply prints the percent character. Note the printed % at the end of Figure 12.8's second sample session.

12.10 Empty Statement

This section supplements the loop material you studied in Chapter 4.

It's sometimes possible to put all of a

loop's functionality inside of its header. For example:

```
   for (int i=0; i<1_000_000_000; i++)
   { }
```

The Java compiler requires that you include a statement for the for loop's body, even if the statement doesn't do anything. The above empty braces ({ }) form a compound statement[2] and satisfy that requirement. In this section, you learn about an alternative way to satisfy that requirement. You learn about the empty statement.

Using the Empty Statement

The *empty statement* consists of a semicolon by itself. Use the empty statement in places where the compiler requires a statement, but there is no need to do anything. For example, the for loop below can be used as a "quick and dirty" way to add a delay to your program:

```
monster.display();
for (int i=0; i<1_000_000_000; i++)
  ;
monster.erase();
```

Coding convention:
Put the empty statement on a line by itself and indent it.

Note how the empty statement is appropriate here because all the work is done in the for loop header, where i counts up to 1 billion. All that counting takes time. Depending on your computer's speed, it might take anywhere from a fraction of a second to 5 seconds.

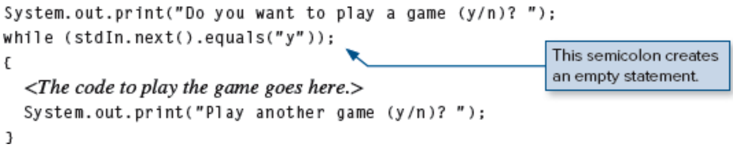
So why would you want to add a delay to your program? Suppose that you're writing a game program that needs to have a monster appear for only a certain time interval. To implement that functionality, print the monster, execute the delay loop, and then erase the monster.

You might want to use the above code fragment as part of a first-cut attempt at implementing the delay, but don't use it for your final implementation. Why? Because it introduces delay that is dependent on the speed of the computer that runs the program. With varied delay, slow computers would have monsters that linger too long, and fast computers would have monsters that disappear too quickly. In a final implementation, you should use the Thread class's sleep method to implement the delay. The sleep method allows you to specify precisely the amount of delay in milliseconds. Because the sleep method might throw an InterruptedException, you need to include throws Exception, as described in Section 3.24 of Chapter 3, or use try and catch, which we explain later in Chapter 15.[3]

In the above code fragment, note the coding-convention callout. Can you think of why it's a good idea to put the empty statement on a line by itself? If you put the empty statement on a line by itself and indent it, readers will see it. On the other hand, if you put the empty statement at the end of the previous statement's line, readers probably won't see it. Seeing the code is an important part of making the code understandable. And making code understandable makes it easier to maintain.

Avoid Accidental Misuse of the Empty Statement

It's fairly common for programmers to create unintended empty statements accidentally. Because you ⚠ enter a semicolon at the end of most lines of Java code, it's easy to get into the habit of hitting the semicolon key at the end of every line of code you write. If you do that at the end of a loop header, it generates an empty statement. Your code might compile and run without a reported error, but it would produce mysterious results. Here is an example:



```
System.out.print("Do you want to play a game (y/n)? ");
while (stdIn.next().equals("y"));
{
  <The code to play the game goes here.>
  System.out.print("Play another game (y/n)? ");
}
```

This semicolon creates an empty statement.

Does the semicolon at the end of the while loop header generate a compilation error? No—the semicolon acts as the lone statement (an empty statement) that's inside the while loop. The subsequent braces form a compound statement. The compound statement is not part of the while loop; it executes after the while loop has finished.

So what does the code do? First, suppose the user enters n. In the while loop header, the JVM compares the entered n value to "y." The loop condition is false, so the JVM skips the while loop's body, the empty statement. The JVM then executes the compound statement and attempts to play a game. That's a logic error: The JVM attempts to play a game even though the user entered n.

Now suppose the user enters y. In the while loop header, the JVM compares the entered y value to “y”. The loop condition is true, so the JVM executes the while loop’s body, the empty statement. The JVM then returns to the loop header and executes the stdIn.next() method call again. The JVM waits for the user to enter another value. But the user won’t know that he or she is supposed to enter anything - because there’s no prompt. That’s a particularly nasty logic error because the program produces no erroneous output and no error message. That means there’s no help in determining what is wrong or what to do about it.

Haste makes waste.

You can produce these same types of logic errors by putting semicolons after “if,” “else if,” or “else” headings. Such semicolons effectively create empty statements, and they’re often introduced accidentally during program development or debugging. Be on the alert for empty statements, and whenever you see one, be suspicious and check it out! Better yet, minimize confusion at the end by maximizing care at the beginning.

12.11 Using break to Exit from a Loop

This section supplements the loop material you studied in Chapter 4. A break statement terminates the immediately enclosing loop and transfers control to the next statement after the bottom of the loop.

The DayTrader program in Figure 12.9 illustrates what's called "day trading." It's a form of gambling in which people buy and sell stock on the stock market every day in hopes of making money off short-term stock movements. This program keeps track of a day trader's stock balance over a three-month period (for day = 1 to 90). The original balance is $1,000. In our simple model, at the beginning of each day, the day trader retains half the initial balance in savings and invests the other half in the stock market. The money returned at the end of the day equals the investment times a random number between 0 and 2. Thus, the money returned ranges anywhere from zero to double the original investment. Each day, the day trader adds the money returned to the balance in savings. If the balance ever goes below $1 or above $5,000, the day trader quits.

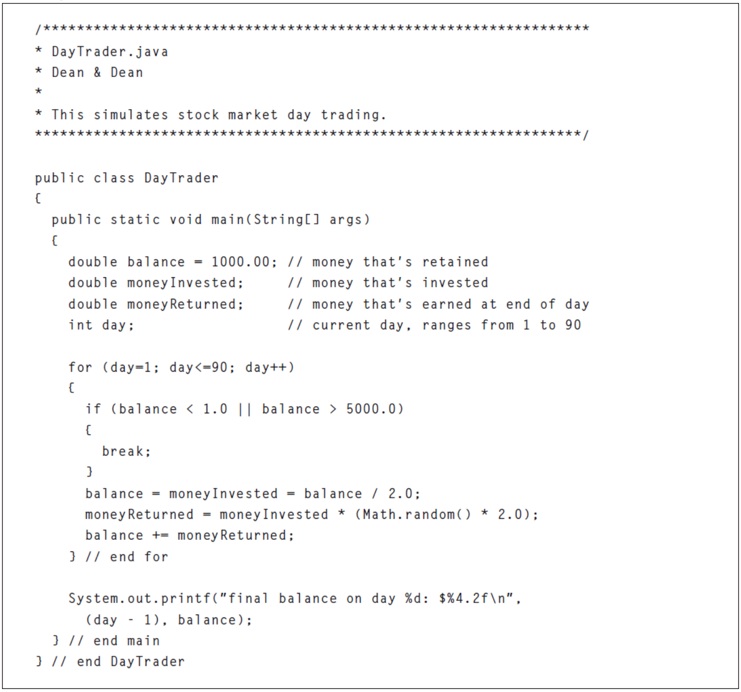
Before examining the break statement in Figure 12.9, look at the (day - 1) argument in the final printf statement. This is after the for loop, so the scope of day needs to be bigger than the scope of the for loop. That's why we declared it before the for loop with the other local variables. But why did we subtract 1 in the printf statement? Because the day++ operation in the third compartment of the for loop header increments day one extra time, after the transaction that drives the balance to a terminating value. If we had forgotten to subtract 1 in the printf statement, that would be an off-by-one error.

Now look at the DayTrader program's break statement. If the balance ever gets outside the $1 to $5,000 range, program control jumps immediately to the next statement below the for loop. If you run the program several times, you'll see that sometimes this causes the loop to terminate before day reaches 90. You'll get a different result each time you run the program because this program uses Math.random to generate a random number in the range between 0.0 and 1.0.

Be aware that you never really have to use a break statement to implement this premature loop-termination capability. For example, you can eliminate the DayTrader program's if and break statements by changing the for loop header to this:

```
for (day=1; day<=90 && !(balance < 1.0 || balance > 5000.0); day++)
```

Don't fall into the trap of using the break statement too often. Usually, someone reading your program will look only at the loop header to figure out how the loop terminates. In using a break statement, you force the reader to look inside the loop for loop termination conditions. And that makes your program harder to understand. Nonetheless, in certain situations, the break statement improves readability rather than hinders it. The DayTrader program's break statement is an example where the break statement improves readability.



```
/*************************************************************
* DayTrader.java
* Dean & Dean
*
* This simulates stock market day trading.
*************************************************************/

public class DayTrader
{
  public static void main(String[] args)
  {
    double balance = 1000.00; // money that's retained
    double moneyInvested;     // money that's invested
    double moneyReturned;     // money that's earned at end of day
    int day;                  // current day, ranges from 1 to 90

    for (day=1; day<=90; day++)
    {
      if (balance < 1.0 || balance > 5000.0)
      {
        break;
      }
      balance = moneyInvested = balance / 2.0;
      moneyReturned = moneyInvested * (Math.random() * 2.0);
      balance += moneyReturned;
    } // end for

    System.out.printf("final balance on day %d: $%4.2f\n",
      (day - 1), balance);
  } // end main
} // end DayTrader
```

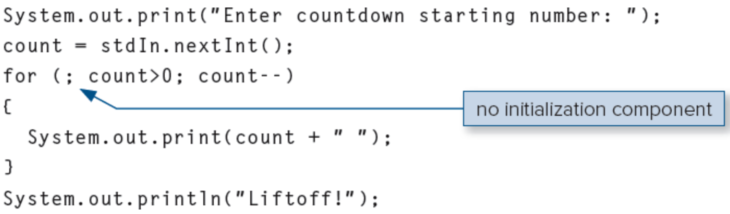
Figure 12.9 DayTrader program, which illustrates use of the break statement

12.12 for Loop Header Details

This section supplements the for loop material that you studied in Chapter 4,

Omitting One or More of the for Loop Header Components

It’s legal, although not all that common, to omit the first and/or third components in the for loop header. For example, to print a countdown from a user-entered number, you could use this code:



```
System.out.print("Enter countdown starting number: ");
count = stdIn.nextInt();
for (; count>0; count--)
{
  System.out.print(count + " ");
}
System.out.println("Liftoff!");
```

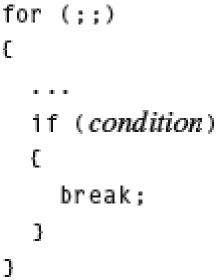
no initialization component

Actually, it’s legal to omit any of the three for loop header components, as long as the two semicolons still appear within the parentheses. For example, you can even write a for loop header like this:

for (;;)

When a for loop header’s condition

component (the second component) is omitted, the condition is considered true for every iteration of the loop. With a permanently true condition, such a loop is often an infinite loop and a logic error. But that's not always the case. You can terminate it by using a break statement like this:

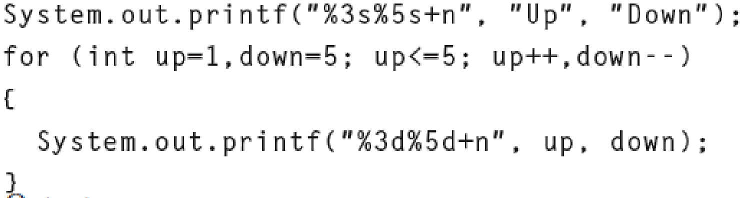


```
for (;;)
{
  ...
  if (condition)
  {
    break;
  }
}
```

You should understand the above example in case you see similar code in someone else's program. But it's rather cryptic, and, as such, you should avoid writing your own code that way.

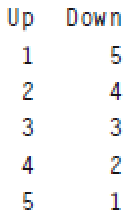
Multiple Initialization and Update Components

For most for loops, one index variable is all that's needed. But every now and then, two or more index variables are needed. To accommodate that need, you can include a list of comma-separated initializations in a for loop header. The caveat for the initializations is that their index variables must be the same type. Working in concert with the comma-separated initializations, you can also include a list of comma- separated updates in a for loop header. The following code fragment and associated output show what we’re talking about. In the for loop header, note the two index variables, up and down, and their comma-separated initialization and update components:



```
System.out.printf("%3s%5s+n", "Up", "Down");
for (int up=1,down=5; up<=5; up++,down--)
{
  System.out.printf("%3d%5d+n", up, down);
}
```

Output:



```
Up  Down
  1    5
  2    4
  3    3
  4    2
  5    1
```

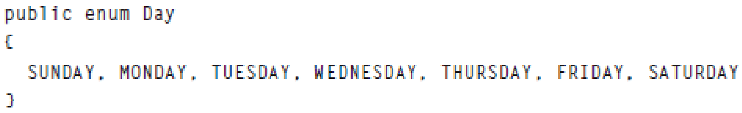
As with many of the techniques presented in this chapter, using multiple initialization and update components in a for loop is a bit of an art. It leads to more compact code, which can be a good thing or a bad thing. If the compact code is more understandable, use it. If the compact code is more cryptic, don't use it.

12.13 Enumerated Types

To declare a primitive variable, you use a primitive type, like int or double. To declare a reference variable, you normally use a class, which serves as a type. Java supports another type, an *enumerated type* (also referred to as an *enum type* or an *enumeration type*), which can be used to restrict a variable to holding one of a fixed set of values defined by the programmer. An enumerated type is sort of like a boolean type, in that a boolean type restricts a variable to holding just the value true or the value false. But with an enumerated type, you specify the name of the enumerated type, and you specify the names of the enumerated type's possible values.

Day of the Week

A classic use for an enumerated type is when you need to keep track of the day of the week. To do that, you could declare a day variable as an int, and use values 0 through 6 to represent Sunday through Saturday. You could get that to work, but the compiler won't stop you from accidentally assigning 7 to day (which would presumably lead to a logic error down the road). A better solution uses the reserved word enum to define a Day enumerated type like this:



```
public enum Day
{
  SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY
}
```

Note that SUNDAY, MONDAY, . . . , SATURDAY use all uppercase. Because an enumerated type's values are named constants, coding conventions dictate that you use uppercase.

After Day is defined, you can use it to declare a day variable like this:

Day day;

And then still later, you can assign one of the enumerated values to day like this:

day = Day.FRIDAY;

This assignment illustrates how you normally need to preface an enumerated value (FRIDAY in this - example) with its enumerated type name (Day in this example).

Letter Grade for a Course

Page 558

Suppose you want to keep track of the letter grades earned in a series of courses taken by a student. You can do that with the help of a Grade enumerated type:

```
private enum Grade {F, D, C, B, A}
```

Note the differences between the Grade enumerated type and the prior Day enumerated type. Day used the public access modifier, presumably because there might be a need for multiple classes to declare variables with the Day type. On the other hand, Grade uses the private access modifier, presumably because only one class (the class that defines Grade) needs to declare variables with the Grade type. Another difference between Day and Grade is that Day uses 4 lines and Grade uses only 1 line. That's simply a style thing. If the enumerated constants take up a lot of space, then put them on a separate line or lines. If they don’t take up much space, then put them on the same line as the enumerated type’s heading.

To see how the Grade type can be used in a complete program, study the GradeManagement program in Figures 12.10a and 12.10b. The GradeManagement program reads in a series of course scores and calculates the letter grade for each course and the grade point average (GPA) for all of the courses combined. Within a loop, the main method calls getGrade to get the letter grade for one course. The getGrade method prompts the user to enter a course’s overall percentage and uses the percentage value to assign the appropriate letter grade enumerated value into a grade variable (which was declared using the Grade enumerated type). If the user enters a negative number (to quit), then grade sticks with its initial value of null. The getGrade method returns grade to main. The main method uses a switch expression to check grade's value and adds 4, 3, 2, 1, or 0 to the totalPts variable, depending on whether grade is an A, B, C, D, or F. When the user enters −1 to quit, main's loop terminates, and main then determines the GPA by dividing totalPts by the number of course percentages entered.

As mentioned earlier, normally you must preface an enumerated type's value with its enumerated type name. So in the GradeManagement program, to assign an A to grade, we use grade = Grade.A;. However, a switch construct's case labels are different. If you want to use an enumerated type's value as a case label, you must use the simple form of the enumerated type's value, such as A, not the full name, Grade.A. To see an example, look for case A: in Figure 12.10a.

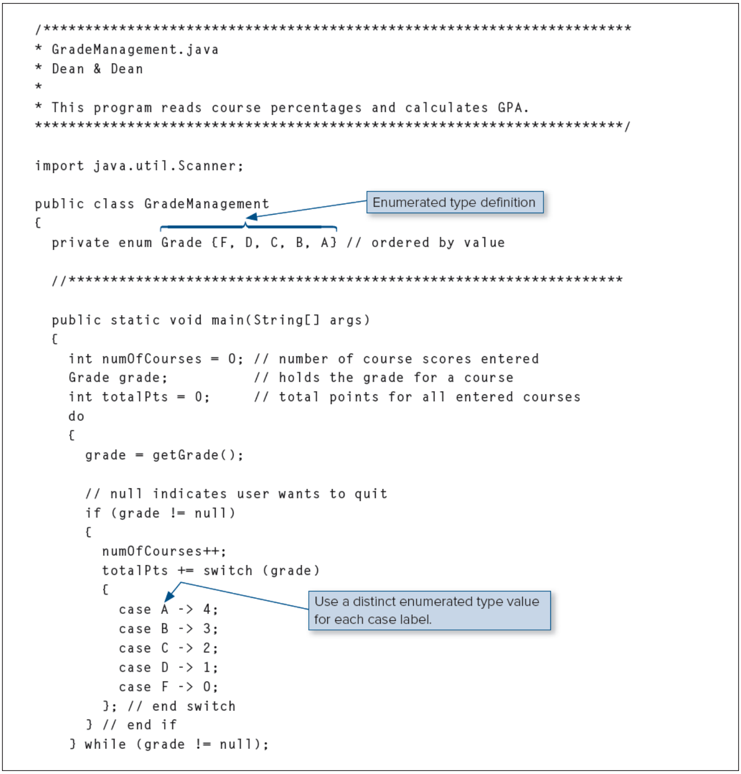
How to Compare Enumerated

Type Values

As you might expect, you can check two enumerated type values for equality or inequality by using the == operator or the != operator. Here’s an example that uses the == operator:

```
if (grade == Grade.F)
{
  System.out.println(
    "If this is a required course, you must retake it.");
}
```

Rather than testing for equality, sometimes you might want to test whether a particular enumerated type value is less than another enumerated type value, where “less than” means that an enumerated type value appears at the left of another enumerated type value in the enumerated type’s original definition. Look at Grade’s enumerated type definition in Figure 12.10a. In the definition, you can see F, D, C, B, and A specified in that order. So you might expect Grade.F < Grade.D to evaluate to true. Unfortunately, that doesn’t work. To compare two enumerated type values with <, >, < =, or > =, you first have to extract the underlying integers from the two enumerated type values. Remember how each char value has an underlying integer value? Similarly, each enumerated type value has an underlying integer value. The leftmost enumerated type value has a 0 associated with it, the second from the left enumerated type value has a 1 associated with it, and so on. To retrieve the integer value, you need to call the ordinal method. Here’s an example:

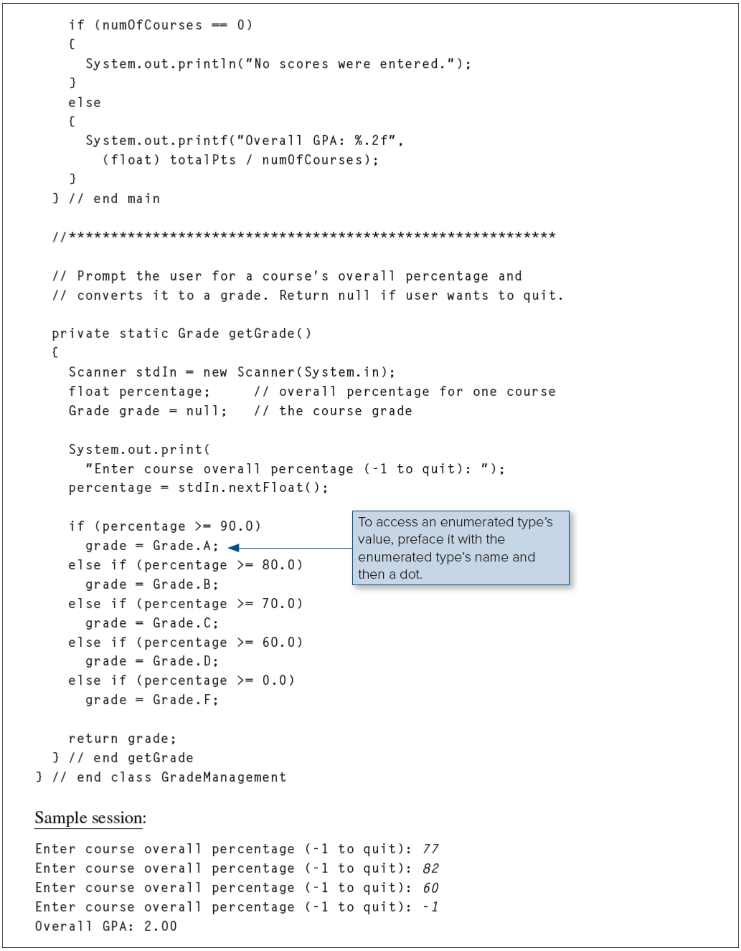


```
/**********************************************************************
* GradeManagement.java
* Dean & Dean
*
* This program reads course percentages and calculates GPA.
**********************************************************************/

import java.util.Scanner;

public class GradeManagement
{
  private enum Grade {F, D, C, B, A} // ordered by value

  //*******************************************************************

  public static void main(String[] args)
  {
    int numOfCourses = 0; // number of course scores entered
    Grade grade;          // holds the grade for a course
    int totalPts = 0;     // total points for all entered courses
    do
    {
      grade = getGrade();

      // null indicates user wants to quit
      if (grade != null)
      {
        numOfCourses++;
        totalPts += switch (grade)
        {
          case A -> 4;
          case B -> 3;
          case C -> 2;
          case D -> 1;
          case F -> 0;
        }; // end switch
      } // end if
    } while (grade != null);
```

Enumerated type definition

Use a distinct enumerated type value for each case label.

Figure 12.10a GradeManagement program—part A



```
    if (numOfCourses == 0)
    {
      System.out.println("No scores were entered.");
    }
    else
    {
      System.out.printf("Overall GPA: %.2f",
        (float) totalPts / numOfCourses);
    }
  } // end main

  //*************************************************************

  // Prompt the user for a course's overall percentage and
  // converts it to a grade. Return null if user wants to quit.

  private static Grade getGrade()
  {
    Scanner stdIn = new Scanner(System.in);
    float percentage;     // overall percentage for one course
    Grade grade = null;   // the course grade

    System.out.print(
      "Enter course overall percentage (-1 to quit): ");
    percentage = stdIn.nextFloat();

    if (percentage >= 90.0)
      grade = Grade.A;
    else if (percentage >= 80.0)
      grade = Grade.B;
    else if (percentage >= 70.0)
      grade = Grade.C;
    else if (percentage >= 60.0)
      grade = Grade.D;
    else if (percentage >= 0.0)
      grade = Grade.F;

    return grade;
  } // end getGrade
} // end class GradeManagement
```

To access an enumerated type's value, preface it with the enumerated type's name and then a dot.

Sample session:

```
Enter course overall percentage (-1 to quit): 77
Enter course overall percentage (-1 to quit): 82
Enter course overall percentage (-1 to quit): 60
Enter course overall percentage (-1 to quit): -1
Overall GPA: 2.00
```

Figure 12.10b GradeManagement program—part B

```
if (grade.ordinal() < Grade.C.ordinal())
{
  System.out.println("If this is a prerequisite course for" +
    " a required course, you must retake it.");
}
```

An Enumerated Type as a Class

In the GradeManagement program, note that the Grade enumerated type is spelled with an uppercase first letter. That coding convention is an indication that an enumerated type is considered to be a class (albeit a special kind of class). Because an enumerated type is a class, you can define it in its own file, separate from other classes. When you save such a file, you'll need to save it with a .java filename extension. And when you compile it, the compiler will generate a .class file. These are all earmarks of a class.

If you'd like to define a fixed list of names that can be used for a variable's value and that's it, then there's no great need to think about an enumerated type as a class. On the other hand, if you'd like to define a fixed list of objects that can be used for a variable's value, then you'll need to know more about the enumerated type's classlike qualities. In this subsection, we describe those qualities.

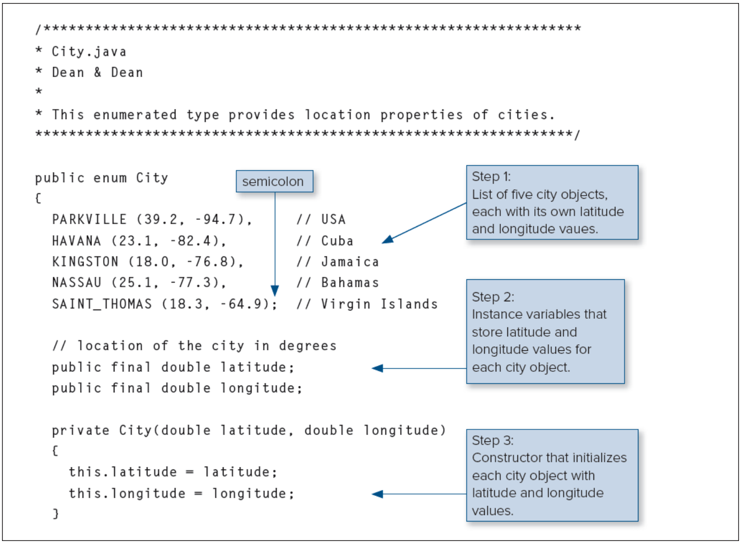
There are three basic steps required to define an enumerated type as a fixed list of objects. It's important to see how to apply the steps to an actual program, so as we explain the steps, please study the step 1, step 2, and step 3 code, as indicated by the callouts in Figure 12.11a.

> **Step 1:** Provide a list of names for the objects, and follow each name with a list of values enclosed in parentheses. Append a semicolon after the last object's parentheses.
>
> **Step 2:** Provide a list of instance variables that are used to store each object's values. Normally, you should use public and final for your instance variable declarations, so the instance variables can be

accessed outside the enumerated type definition without any risk of their being changed.

Step 3: Provide a private constructor that's used to initialize each of the object's instance variables. The compiler requires the constructor to be private, so it's impossible to call the enumerated type constructor (and create new enumerated type objects) from outside the enumerated type definition. If no external constructor calls are allowed, where is the constructor called from? From inside the enumerated type's definition! As described in Step 1, an enumerated type definition contains a list of object names with parentheses after each name. Each of those list items is a constructor call. For example, in Figure 12.11a's City enumerated type definition, PARKVILLE (39.2, -94.7) calls City's constructor and passes 39.2 to the latitude parameter and −94.7 to the longitude parameter.

If you'd like to manipulate an enumerated type's values, you can use a driver program to retrieve the values and manipulate them from within the driver. On the other hand, if you think the manipulation is general-purpose in nature, such that more than one program might find it useful, then you should consider implementing the manipulation as a public method inside the enumerated type definition. For example, see the getDistance method in Figure 12.11b. It calculates the distance between two City objects—the City object that calls getDistance and the City object that's passed in as an argument to getDistance. Lots of programs might need to know the distance between cities, so we chose to put the method within the City enumerated type definition, rather than within a driver class.



```
/*********************************************************************
* City.java
* Dean & Dean
*
* This enumerated type provides location properties of cities.
*********************************************************************/

public enum City
{
  PARKVILLE (39.2, -94.7),        // USA
  HAVANA (23.1, -82.4),           // Cuba
  KINGSTON (18.0, -76.8),         // Jamaica
  NASSAU (25.1, -77.3),           // Bahamas
  SAINT_THOMAS (18.3, -64.9);     // Virgin Islands

  // location of the city in degrees
  public final double latitude;
  public final double longitude;

  private City(double latitude, double longitude)
  {
    this.latitude = latitude;
    this.longitude = longitude;
  }
```

semicolon

Step 1:
List of five city objects, each with its own latitude and longitude vaues.

Step 2:
Instance variables that store latitude and longitude values for each city object.

Step 3:
Constructor that initializes each city object with latitude and longitude values.

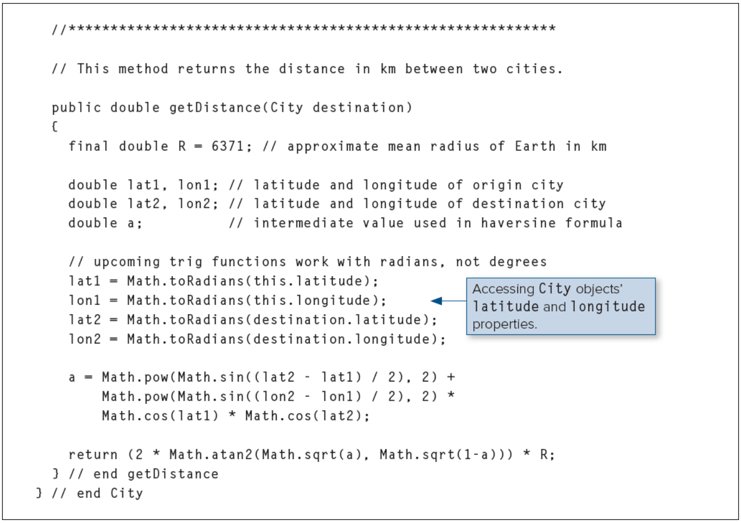
Figure 12.11a City enumerated type—part A

Page 562

Although the getDistance method appears within an enumerated type definition, its syntax is identical to methods defined within a regular class. To illustrate that point, note how the latitude value is accessed with either the calling object (this.latitude) or the parameter object (destination.latitude). That should look familiar because accessing a value within an enumerated type object is just like accessing any other object value—use the dot operator.

The getDistance method's algorithm for finding the *geodesic distance* (the shortest distance between two points on a curved surface) is rather involved. It uses the cities' latitude and longitude values as inputs to the haversine formula. An explanation of the haversine formula is beyond the scope of this book, but if you're curious, Google it. On the other hand, considering the ubiquity of the global positioning system (GPS), everyone should know a little about latitude and longitude, so here goes . . .

Latitude values start at zero degrees (0°) at the equator. As you go north from there, degrees go from 0° to positive 90°. As you go south, degrees go from 0° to negative 90°. The North Pole is at 90°, and the South Pole is at −90°. Longitude values start at 0° at the Prime Meridian, which passes through the Royal Observatory in Greenwich, England. As you go east from there, degrees go in a positive direction from 0° to 180°. As you go west from the Prime Meridian, degrees go in a negative direction from 0° to −180°. The positive and negative 180° longitudes meet at the Anti-Meridian, otherwise known as the International Date Line. The International Date Line is the first place where each new day begins. It primarily passes through open waters in the Pacific Ocean, but it also crosses parts of Russia, Fiji, and Antarctica.



```
  //**********************************************************

  // This method returns the distance in km between two cities.

  public double getDistance(City destination)
  {
    final double R = 6371; // approximate mean radius of Earth in km

    double lat1, lon1; // latitude and longitude of origin city
    double lat2, lon2; // latitude and longitude of destination city
    double a;          // intermediate value used in haversine formula

    // upcoming trig functions work with radians, not degrees
    lat1 = Math.toRadians(this.latitude);
    lon1 = Math.toRadians(this.longitude);
    lat2 = Math.toRadians(destination.latitude);
    lon2 = Math.toRadians(destination.longitude);

    a = Math.pow(Math.sin((lat2 - lat1) / 2), 2) +
        Math.pow(Math.sin((lon2 - lon1) / 2), 2) *
        Math.cos(lat1) * Math.cos(lat2);

    return (2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a))) * R;
  } // end getDistance
} // end City
```

Accessing `City` objects' `latitude` and `longitude` properties.

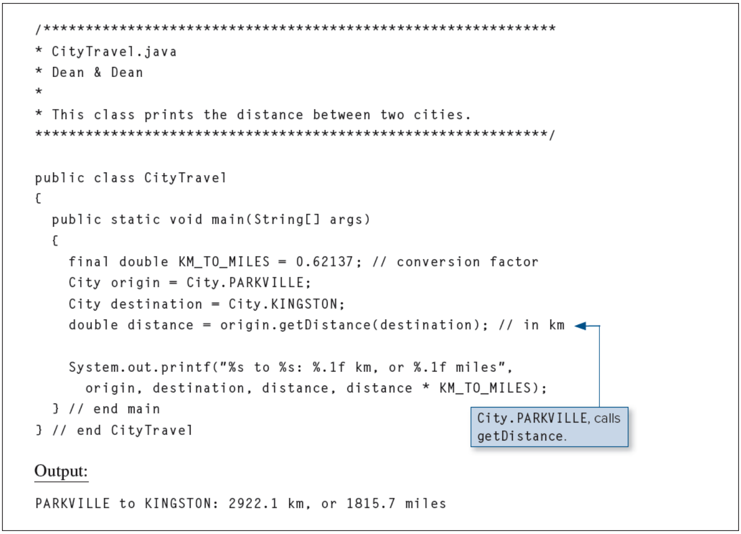
Figure 12.11b City enumerated type—part B

Now look at how a driver program can use the City enumerated type. Figure 12.12's CityTravel program determines the distance between Parkville, Missouri, and Kingston, Jamaica—two of the cities in the City enumerated type. The program was designed to help Park University's fledgling Oceanography and Computer Science Department. For lab classes, the department flies its students from its private airfield in Parkville to various island cities in the West Indies.

Retrieving All the Objects in an Enumerated Type

Sometimes you might have a need to retrieve all the objects in an enumerated type. You can do that by calling the enumerated type's values method.[4] It generates an array of the enumerated type's objects. You can then process each of the generated array's objects using a standard loop or a for-each loop. For example, here's how you can retrieve the names of all the City objects from the City enumerated type and print them:

```
for (City city : City.values())
{
  System.out.print(city + " ");
}
```



```
/*************************************************************
* CityTravel.java
* Dean & Dean
*
* This class prints the distance between two cities.
*************************************************************/

public class CityTravel
{
  public static void main(String[] args)
  {
    final double KM_TO_MILES = 0.62137; // conversion factor
    City origin = City.PARKVILLE;
    City destination = City.KINGSTON;
    double distance = origin.getDistance(destination); // in km

    System.out.printf("%s to %s: %.1f km, or %.1f miles",
      origin, destination, distance, distance * KM_TO_MILES);
  } // end main
} // end CityTravel
```

City.PARKVILLE, calls getDistance.

Output:

```
PARKVILLE to KINGSTON: 2922.1 km, or 1815.7 miles
```

Figure 12.12 CityTravel program

Page 564

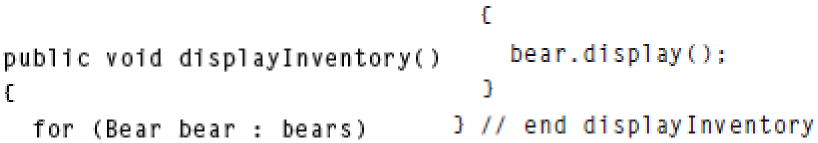
In the CityTravel program, the main method uses hardcoded values (City.PARKVILLE and City.KINGSTON) to specify the origin and destination cities for a particular distance calculation. In an exercise at the end of this chapter, you’re asked to improve that code by prompting the user to enter origin and destination cities and using the user-entered values in the getDistance method call. If you choose to do the exercise, you’ll find you’ll need to loop through all of the cities in the City enumerated type. You can use the code fragment above as a starting point.

12.14 forEach Method, Lambda Expressions, Method References, and Streams

Simple OOP asks old dogs to do tricks they already know. Now, we'll begin asking those dogs to do other tricks by teaching them new tricks or pointing to other tricks that other animals also could do.

forEach Method and Its Consumer Argument

Consider the BearStore program presented in Section 10.4. In Figure 10.5b, the BearStore class's -displayInventory method looks like this:



```
public void displayInventory()
{
  for (Bear bear : bears)
  {
    bear.display();
  }
} // end displayInventory
```

Because bears is an ArrayList, we can replace this method's for-each loop with a forEach method call. The forEach method takes a single argument. This argument is not a primitive or even an object. It is a method that implements the Consumer functional interface. A *functional interface* is an interface that contains only one *abstract method*, which is a method signature with no method body. In the context of a functional interface, the abstract method is known as a *functional method*. The Consumer functional interface defines an accept functional method with an Object parameter:

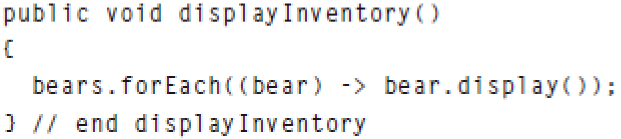
void accept(Object object)

In implementing a *functional*

interface, the implementing method must define a method whose signature is the same as that of the functional interface's functional method. Thus, a method that implements the Consumer interface must have one parameter, and that parameter must be an Object. We'll have more to say about interfaces in Chapter 14.

The question is, how can forEach's argument be a method (to match the argument's type, which is a Consumer functional interface)? There are two techniques: (1) We can anonymously define the method with a *lambda expression*, or (2) we can identify an already written external method with a *method reference*. The following discussion provides an introduction to lambda expressions and method references in non-GUI contexts, You'll see more of these two constructs as GUI event handlers in Chapters 17, 18, and 19.

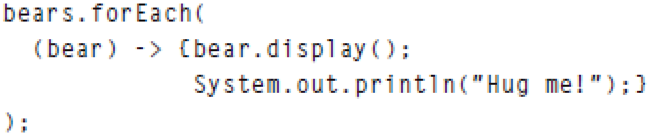
Here’s the code for a BearStore2 displayInventory method, which uses a forEach method call to iterate through the elements of the bears ArrayList. This forEach argument implements the Consumer interface with a lambda expression.



```
public void displayInventory()
{
  bears.forEach((bear) -> bear.display());
} // end displayInventory
```

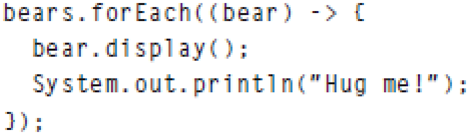
A lambda expression is odd looking at first, but when you think about it, its meaning becomes clear. It says “for each bear element, call its display method.”

Notice that we did not terminate the lambda expression’s bear.display() method call with a semicolon. This is appropriate when the Consumer implementation of the forEach argument has only one statement. If the Consumer implementation of the forEach argument has more than one statement, we terminate each statement with a semicolon and enclose them all in a pair of braces, like this:



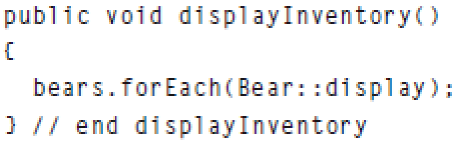
```
bears.forEach(
  (bear) -> {bear.display();
             System.out.println("Hug me!");}
);
```

Above, note how the lambda expression’s statements are indented. That works fine when the statements are relatively short, but when they’re longer, that will lead to line wrap. To avoid that situation, here’s an alternative indentation style for lambda expressions:



```
bears.forEach((bear) -> {
  bear.display();
  System.out.println("Hug me!");
});
```

Because Figure 10.4’s Bear class already has a method that fulfills our original intent (but not the above “Hug me!” variation), as an alternative, we can implement the Consumer argument of the forEach version of the original program with a method reference, like this:



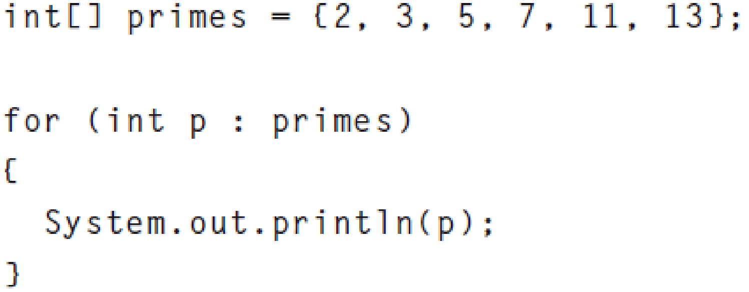
```
public void displayInventory()
{
  bears.forEach(Bear::display);
} // end displayInventory
```

Page 566

In this code’s forEach argument, the double colon separates a class name on the left from a method name on the right. The class name indicates where the method is located. The referenced method may be any conforming method in the Java API (provided the current program implicitly or explicitly imports its package), or it may be a method you define. If the referenced method is in the same class as the forEach method that invokes it, the class identifier before the double colon in the forEach argument may be the simple reserved word, this.

When a Consumer implementation is complex and/or you want to invoke it more than once with multiple forEach method calls, instead of using a lambda expression, it’s better to put that implementation in a separate method and access it with a method reference.

Now let’s consider another example, which initially seems simpler but actually is more complex. In Section 9.11, we presented a code fragment with a for-each loop that stepped through and printed the elements of an array if int primitives. Here again is that code fragment:



```
int[] primes = {2, 3, 5, 7, 11, 13};


for (int p : primes)
{
  System.out.println(p);
}
```

If we try to replace this code’s for-each loop with a corresponding forEach method call, we stumble over two impediments: (1) Although Iterable collections like ArrayList provide forEach methods, an ordinary array does not. (2) For the method that implements the Consumer interface, its parameter must be an object, not a primitive.

To convert the code fragment’s primes array into a List, which provides a forEach method because it implements the Iterable interface, we add this import statement:

```
import java.util.Arrays;
```

To convert our code fragment’s array of primitives into a corresponding array of Integer objects, we change our code fragment’s declaration to this:

```
Integer[] primes = {2, 3, 5, 7, 11, 13};
```

Now we can replace our code fragment’s for-each loop with just one statement that uses a forEach method whose argument is a lambda expression:

```
Arrays.asList(primes).forEach((p) -> System.out.println(p));
```

Alternatively, we can replace our code

fragment's for-each loop with one statement that uses a forEach method whose argument is a special method reference:

```
Arrays.asList(primes).forEach(System.out::
```

The System.out to the left of the double colon is the System class name followed by one of its fields, out, which refers to the Java API class, PrintStream. The println at the right of the double colon identifies one of PrintStream's methods.

Most of us would not choose to use forEach to step through an array of primitives because then forEach forces us to convert the array of primitives to a collection of objects, where the collection (e.g., a List) implements the Iterable interface. However, the forEach method is attractive when the initial structure is a class that implements the Iterable interface.

That's any class that implements the -Collection interface. (See Figure 10.15.)

Streams

Oracle's Java documentation describes a Java stream as: "A sequence of elements supporting sequential and parallel aggregate operations." Often a stream can produce the same result(s) that an array or collection could produce. But a stream is not a data structure. It's a collection of processes that operate on a data structure. For example, Figure 12.13 shows a StockAverage2 program, which uses a Stream to process data entered into an ArrayList.

Except for omission of a stockSum declaration, the StockAverage2 program in Figure 12.13 is like the StockAverage program in Figure 10.3 before the while loop. After the while loop in the previous StockAverage program, a for loop computes a value for stockSum and then an if statement computes and prints an average stock value if the size of the ArrayList is not zero. After the while loop in the current StockAverage2 program, there is just one statement—the print statement. This print statement differs from the previous program's print statement only in the nature of its printed value.

The StockAverage program's printed value was the computed average, stockSum / stocks.size(). The current program's printed value is a stocks.stream() method call that converts the ArrayList to a Stream object, followed by this chain of Stream method calls:

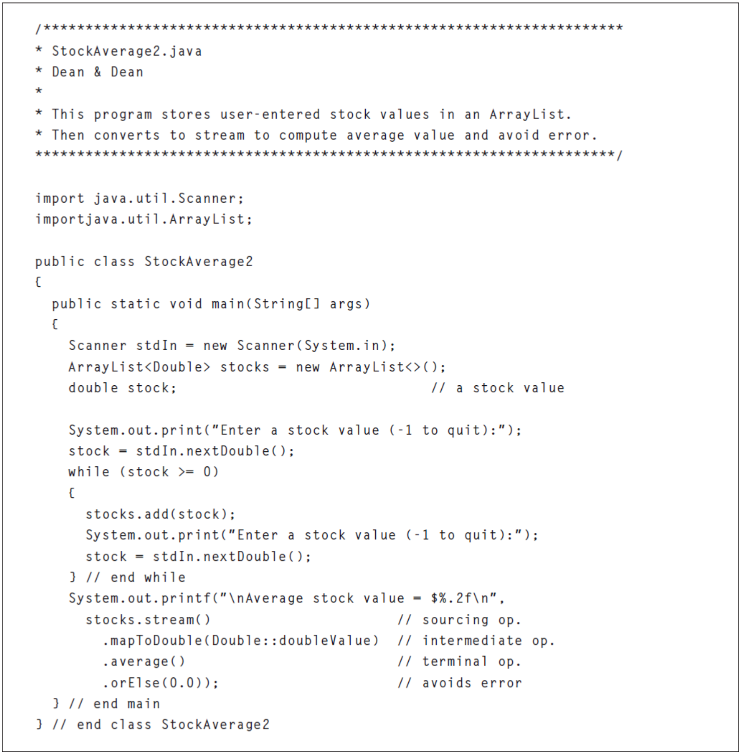
```
.mapToDouble(Double::doubleValue).avera
```

The first of these uses a method reference to convert Stream elements from Double to double and returns a

DoubleStream of double elements. The second computes the average of the double elements in the DoubleStream and returns an OptionalDouble object. OptionalDouble's orElse method does the job of the StockAverage program's if statement. Normally, it just returns the average value, but if the length of the ArrayList source is zero, the average of its unboxed elements is indeterminate, and the orElse(0.0) method returns its own argument's value (0.0) instead.

Notice that the StockAverage2 program is significantly shorter than the StockAverage program in Figure 10.3. That's because the chained stream operations in the StockAverage2 program's print statement replaces ten lines of code in the for loop and if statement at the end of the StockAverage program.5 An important feature of Java streams is the large number of data processing alternatives they provide.

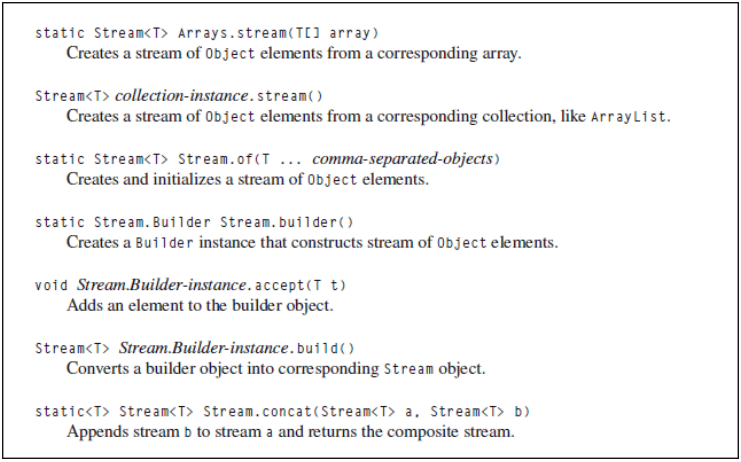
When you execute a Java program that contains one or more related streams, the computer organizes these streams into a bundle of code called a *pipeline*, which "lays out a sequence of operations that pass a collection of items between them."6 Each operation in the sequence is either *sourcing*, *intermediate*, or *terminal*. The StockAverage2 program shows one example of each of these. A pipeline can have only one terminal operation, and nothing happens until that terminal operation occurs. Pipeline code reads its source data all at once at the very last moment. Although normal code can capture the result returned by a pipeline's terminal operation and save it in a normal variable, array, or collection, which could be the same as the pipeline's source, the pipeline code itself cannot alter its source's data. The pipeline's terminal operation automatically deletes everything in the pipeline, preventing its reuse. Thus, a stream pipeline is less intrusive and more ephemeral than other Java code.



```
/*********************************************************************
* StockAverage2.java
* Dean & Dean
*
* This program stores user-entered stock values in an ArrayList.
* Then converts to stream to compute average value and avoid error.
*********************************************************************/

import java.util.Scanner;
importjava.util.ArrayList;

public class StockAverage2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ArrayList<Double> stocks = new ArrayList<>();
    double stock;                            // a stock value

    System.out.print(”Enter a stock value (-1 to quit):”);
    stock = stdIn.nextDouble();
    while (stock >= 0)
    {
      stocks.add(stock);
      System.out.print(”Enter a stock value (-1 to quit):”);
      stock = stdIn.nextDouble();
    } // end while
    System.out.printf(”\nAverage stock value = $%.2f\n”,
      stocks.stream()                    // sourcing op.
        .mapToDouble(Double::doubleValue)  // intermediate op.
        .average()                       // terminal op.
        .orElse(0.0));                   // avoids error
  } // end main
} // end class StockAverage2
```

Figure 12.13 StockAverage2 program that mimics the StockAverage program in Figure 10.3 by using a Stream to process ArrayList data.

Figure 12.14a identifies several stream sourcing methods. These create streams in several different ways, including conversion from existing arrays and collections, initialization with a comma-separated sequence of objects, progressive accumulation by a Stream.Builder object, and concatenation of previously existing streams.



`static Stream<T> Arrays.stream(T[] array)`
Creates a stream of `Object` elements from a corresponding array.

`Stream<T>` *collection-instance*`.stream()`
Creates a stream of `Object` elements from a corresponding collection, like `ArrayList`.

`static Stream<T> Stream.of(T ...` *comma-separated-objects*`)`
Creates and initializes a stream of `Object` elements.

`static Stream.Builder Stream.builder()`
Creates a `Builder` instance that constructs stream of `Object` elements.

`void` *Stream.Builder-instance*`.accept(T t)`
Adds an element to the builder object.

`Stream<T>` *Stream.Builder-instance*`.build()`
Converts a builder object into corresponding `Stream` object.

`static<T> Stream<T> Stream.concat(Stream<T> a, Stream<T> b)`
Appends stream `b` to stream `a` and returns the composite stream.

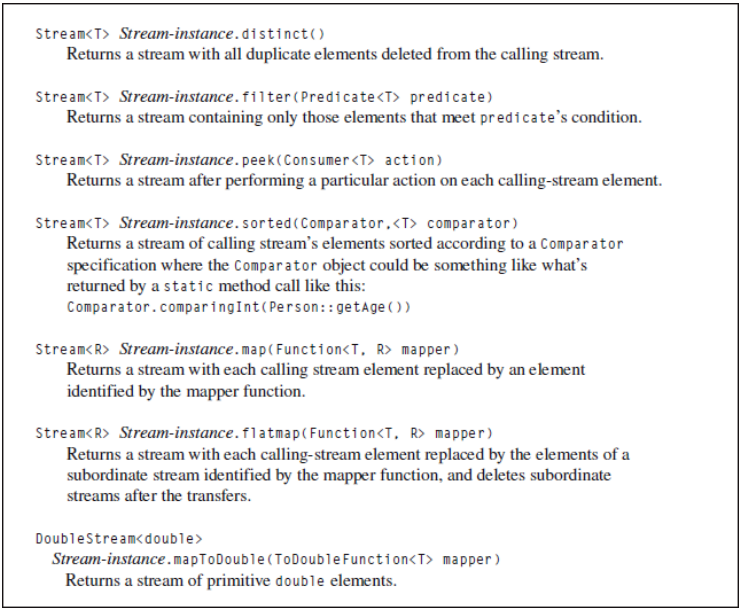
Figure 12.14a Some alternative Stream sourcing operations The Methods shown are for Object streams. Except for *collection-instance*.stream() The IntegerStream class provides similar methods for int streams, and the DoubleStream class provides similar

methods for double streams.

If you create a stream from an array or collection, remember that the stream and its source are separate objects. You can modify the stream and its elements in a wide variety of ways. However, unless you explicitly replace a sourcing array or collection with what is returned by a stream's terminal operation, when you terminate the stream, any changes you make to the stream do not alter the stream's source.

Figure 12.14b identifies several intermediate methods. For easy chaining, each of these returns another stream instance. The returned instance is typically different from the calling instance, but it may be exactly the same. For example, the peek method may simply generate side effects like printing when particular conditions occur. Thus, you can use peek for stream debugging. For example, in the StockAverages2 program in Figure

12.13, we could verify the capture of user entries by inserting .peek(System.out::println) either before or after the stream's mapToDouble method call.



`Stream<T>` *Stream-instance*`.distinct()`
Returns a stream with all duplicate elements deleted from the calling stream.

`Stream<T>` *Stream-instance*`.filter(Predicate<T> predicate)`
Returns a stream containing only those elements that meet `predicate`'s condition.

`Stream<T>` *Stream-instance*`.peek(Consumer<T> action)`
Returns a stream after performing a particular action on each calling-stream element.

`Stream<T>` *Stream-instance*`.sorted(Comparator,<T> comparator)`
Returns a stream of calling stream's elements sorted according to a `Comparator` specification where the `Comparator` object could be something like what's returned by a `static` method call like this:
`Comparator.comparingInt(Person::getAge())`

`Stream<R>` *Stream-instance*`.map(Function<T, R> mapper)`
Returns a stream with each calling stream element replaced by an element identified by the mapper function.

`Stream<R>` *Stream-instance*`.flatmap(Function<T, R> mapper)`
Returns a stream with each calling-stream element replaced by the elements of a subordinate stream identified by the mapper function, and deletes subordinate streams after the transfers.

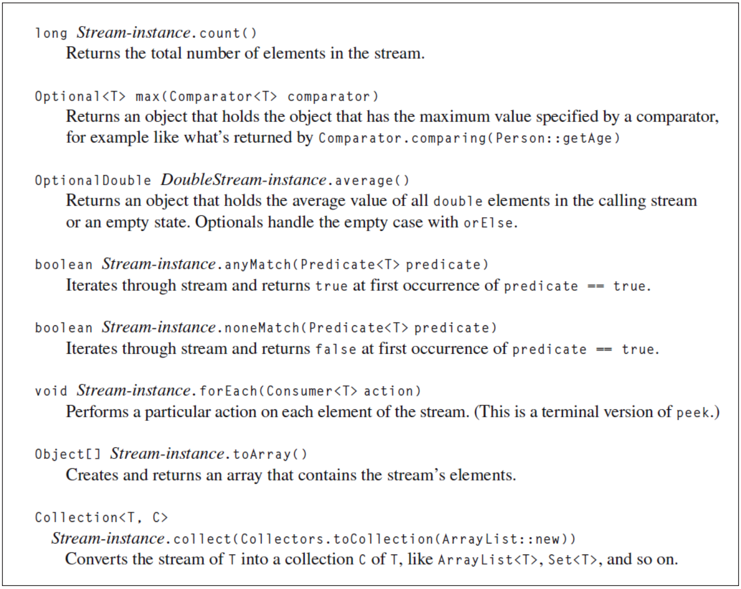
`DoubleStream<double>`
Stream-instance`.mapToDouble(ToDoubleFunction<T> mapper)`
Returns a stream of primitive `double` elements.

Figure 12.14b Selected Stream intermediate operations Predicate, Consumer, Comparator, Function, and ToDoubleFunction are functional interfaces, which you can implement with a method defined in a local lambda expression or identified by a method reference.

As indicated in Figure 12.14b, peek's

argument must implement the Consumer interface, and mapToDouble's argument must implement the ToDoubleFunction interface. Oracle documentation identifies Consumer and ToDoubleFunction as functional interfaces.

To make a stream's operations execute, you must explicitly terminate a stream with one of its terminal methods, like those in Figure 12.14c. It's possible to create a stream variable and update it at different places in your code. But frequently you'll want to combine a stream's creation, processing, and termination in a chained statement like that in our StockAverages2 program. In either case, the final stream operation should be a terminal method call like one of those in Figure 12.14c. Note that the terminal stream operation in the StockAverage2 program is the average method call. The orElse at the end of StockAverage2's method-call chain is a nonstream method called by the OptionalDouble object, which the terminal stream method returned.



`long` *Stream-instance*`.count()`
Returns the total number of elements in the stream.

`Optional<T> max(Comparator<T> comparator)`
Returns an object that holds the object that has the maximum value specified by a comparator, for example like what's returned by `Comparator.comparing(Person::getAge)`

`OptionalDouble` *DoubleStream-instance*`.average()`
Returns an object that holds the average value of all `double` elements in the calling stream or an empty state. Optionals handle the empty case with `orElse`.

`boolean` *Stream-instance*`.anyMatch(Predicate<T> predicate)`
Iterates through stream and returns `true` at first occurrence of `predicate == true`.

`boolean` *Stream-instance*`.noneMatch(Predicate<T> predicate)`
Iterates through stream and returns `false` at first occurrence of `predicate == true`.

`void` *Stream-instance*`.forEach(Consumer<T> action)`
Performs a particular action on each element of the stream. (This is a terminal version of `peek`.)

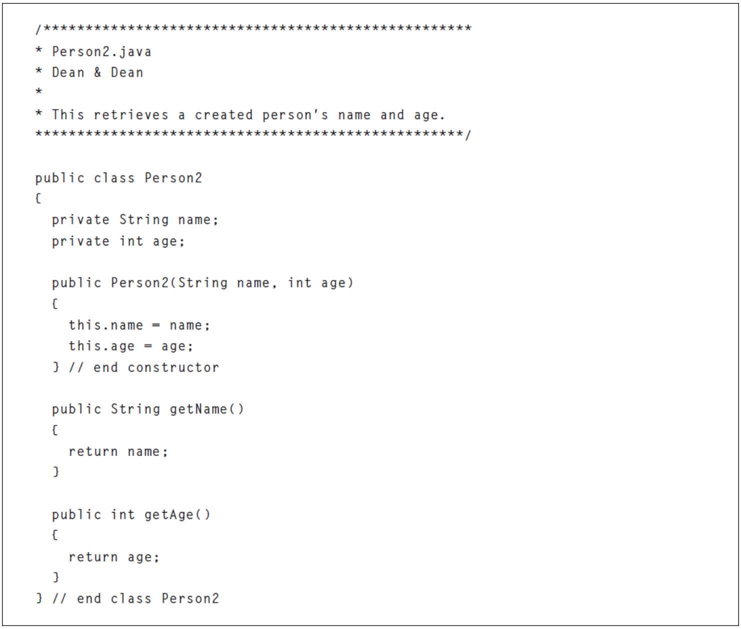
`Object[]` *Stream-instance*`.toArray()`
Creates and returns an array that contains the stream's elements.

`Collection<T, C>`
Stream-instance`.collect(Collectors.toCollection(ArrayList::new))`
Converts the stream of `T` into a collection `C` of `T`, like `ArrayList<T>`, `Set<T>`, and so on.

Figure 12.14c Several Stream terminal operations

Some terminal methods generate a simple result like the current element count. Others return a variation of Optional, which can handle empty cases gracefully. Other terminal methods return a boolean that indicates whether all or any elements satisfy a certain condition. Still other terminal methods return a more permanent version of the stream, like an array or any of the many types of Java collections. The result could become a completely new object, or it could be used to replace an array or collection that was the source of the original stream. For a terminal method that does nothing except cause all previous stream methods to execute, use: forEach(e -> {}). This forEach argument's lambda expression specifies an anonymous method that does nothing.

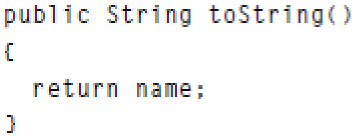
To show how a Comparator works and to see a nontrivial lambda expression in action, we'll use a variation of Figure 7.7's Person class—the Person2 class in Figure 12.15a.



```
/*******************************************************
* Person2.java
* Dean & Dean
*
* This retrieves a created person's name and age.
*******************************************************/

public class Person2
{
  private String name;
  private int age;

  public Person2(String name, int age)
  {
    this.name = name;
    this.age = age;
  } // end constructor

  public String getName()
  {
    return name;
  }

  public int getAge()
  {
    return age;
  }
} // end class Person2
```

Figure 12.15a Person2 class, used by the Person2Driver class in Figure 12.15b The Person2 class is a variation of the Person class in Figure 7.7.

We drive the Person2 class with the Person2Driver class in Figure 12.15b. In Person2Driver, for peek's argument we use a lambda expression because Person2 does not provide a default string. Now suppose we go back to the Person2 class and add the following method:

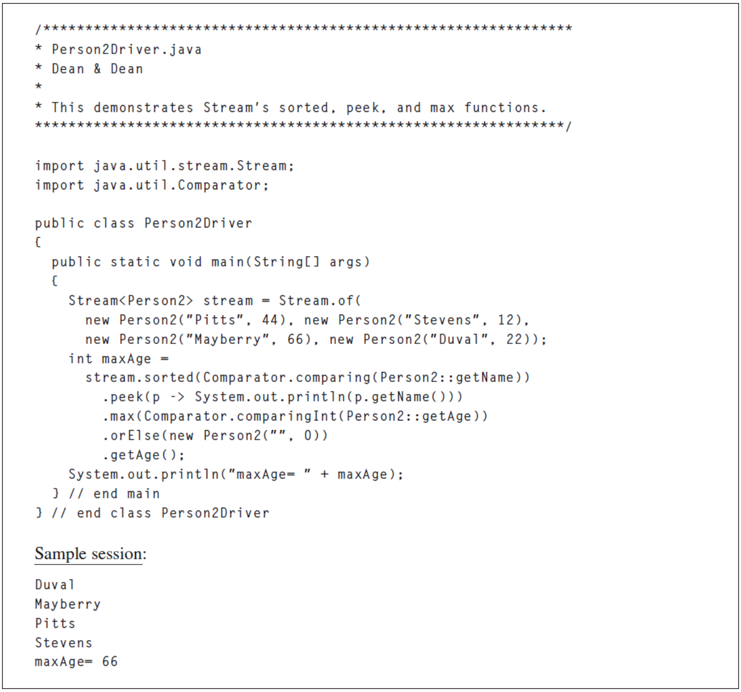


```
public String toString()
{
  return name;
}
```

Then, in Person2Driver, peek's argument also could be a method reference, like this:

.peek(System.out::println)

That works, because (as you might recall), if there's an attempt to print a reference variable with a println statement (or print or printf), the reference variable's toString method is invoked automatically.



```
/*******************************************************************
* Person2Driver.java
* Dean & Dean
*
* This demonstrates Stream’s sorted, peek, and max functions.
*******************************************************************/

import java.util.stream.Stream;
import java.util.Comparator;

public class Person2Driver
{
  public static void main(String[] args)
  {
    Stream<Person2> stream = Stream.of(
      new Person2(”Pitts”, 44), new Person2(”Stevens”, 12),
      new Person2(”Mayberry”, 66), new Person2(”Duval”, 22));
    int maxAge =
      stream.sorted(Comparator.comparing(Person2::getName))
        .peek(p -> System.out.println(p.getName()))
        .max(Comparator.comparingInt(Person2::getAge))
        .orElse(new Person2(””, 0))
        .getAge();
    System.out.println(”maxAge= ” + maxAge);
  } // end main
} // end class Person2Driver
```

Sample session:

```
Duval
Mayberry
Pitts
Stevens
maxAge= 66
```

Figure 12.15b Person2Driver class, which uses Person2 class in Figure 12.15a This illustrates use of Comparator and both kinds of functional-interface implementation.

12.15 Hexadecimal, Octal, and Binary Numbers

Normal numbers are expressed as powers of 10, but because computers are binary and 16 is a simple power of 2 ($16 = 2^4$), it's common practice to express computer quantities in base 16 (using powers of 16), rather than base 10 (using powers of 10). Base 10 numbers are called decimal numbers. Base 16 numbers are called *hexadecimal* numbers. The places in decimal numbers are called *digits*. The places in hexadecimal numbers are sometimes called *hexits*, but more often, they're simply called *hexadecimal digits*. Base 10 numbers use the 10 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Base 16 numbers use the 16 symbols: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, and f (uppercase letters A through F are equivalent to a through f). Thus, hexadecimal numbers frequently include one or more of the first six alphabetic characters, as well as one or more of the normal numerical characters. In addition to base 10, base 2, and base 16, computer quantities are sometimes represented with base 8. Base 8 is referred to as *octal*. As you'd expect, octal uses the eight symbols 0 through 7.

In Java, you can write any integer in decimal, hexadecimal, binary, or octal form. If you want a number to be interpreted as hexadecimal, you must prefix it with zero followed by x or X; that is, with 0x or 0X. If you want a number to be interpreted as binary, you must prefix it with zero followed by b or B; that is, 0b or 0B. There is no corresponding prefix for octal. If you want a number to be interpreted as octal, you must prefix it with a zero (and no additional character). So for example, if you see 0x263A, you can recognize it as a hexadecimal number. Similarly, if you see 0b01010111, you can recognize it as a binary number.

And if you see 042, you can recognize it as an octal number. For most of us, hexadecimal, octal, and binary numbers are not very intuitive. However, if you can write the number you want to convert (i.e., decimal, hexadecimal, octal, or binary), it's pretty easy to make conversions. Just use Integer's two-parameter toString method:

Integer.toString(*starting-number, desired-base*);

The starting number is the number you want to convert.

For example, if you want to see the decimal equivalent of 0x263A, use this:

```
System.out.println(Integer.toString(0x263A
10));
```

This generates an output of 9786. To go the other way, you could use the formatted print statement

```
System.out.printf("%x\n", 9786);
```

This generates an output of 263a. Notice that this method’s output does not include the 0x prefix, and it uses lowercase letters for the alphabetic hexadecimal digits.

Similarly, you can use printf’s conversion specifiers to print the hexadecimal, decimal, or octal equivalents of hexadecimal, decimal, or binary numbers. Thus, you could print the hexadecimal, decimal, and octal equivalents of the binary number 0b01010111 with these statements:

```
System.out.printf("%x\n", 0b01010111);
System.out.printf("%d\n", 0b01010111);
System.out.printf("%o\n", 0b01010111);
```

These generate the outputs, 57, 87, and 127, respectively.

Conversely, if you want to see the hexadecimal, octal, and binary equivalents of the decimal number 9786, you could use these three print statements:

```
System.out.println(Integer.toString(9786, 16));
System.out.println(Integer.toString(9786, 8));
System.out.println(Integer.toString(9786, 2));
```

These generate the outputs, 263a, 23072, and 10011000111010, respectively.

12.16 GUI Track: Unicode (Optional)

Earlier, you learned that characters get their underlying numeric values from the ASCII character set. That’s true for the 128 characters in ASCII, but there are way more than 128 characters in the world. The ASCII character set contains the characters in the Latin alphabet—A through Z—but it does not contain the characters in other alphabets. For example, it does not contain the characters in the Greek, Cyrillic, and Hebrew alphabets. The designers of the Java language wanted Java to be general purpose, so they wanted to be able to produce text output for many different languages using many different alphabets. To handle the additional characters, the Java designers had to use a bigger character set than the ASCII character set. Thus, they adopted the *Unicode* standard. The Unicode standard defines underlying numeric values for a huge set of 65,536 characters.

Why are there 65,536 characters in the Unicode standard? Because the people who designed the Unicode standard (the Unicode Consortium) decided that 16 bits would be sufficient to represent all the characters needed in a computer program.7 And 16 bits can represent 65,536 characters. Here are the binary representations for the first four characters and the last character:

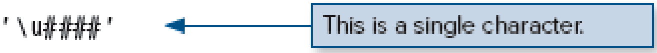
```
0000 0000 0000 0000
0000 0000 0000 0001
0000 0000 0000 0010
0000 0000 0000 0011
. . .
1111 1111 1111 1111
```

Notice that each row is a different permutation of 0’s and 1’s. If you wrote all such permutations, you’d see 65,536 rows. Thus, with 16 bits, you can represent 65,536 characters. The formula for determining the number of permutations (and consequently the number of rows and the number of characters) is 2 raised to the power of the number of bits. In other words, $2^{16} = 65,536$. You can apply that same reasoning to understand why there are 128 characters in the ASCII character set. Way back in 1963 (when dinosaurs and punched cards roamed the Earth), the people who designed the ASCII character set decided that 7 bits would be sufficient to represent all the characters needed in a computer program. $2^7 = 128$, so 7 bits can represent 128 unique values.

Because the ASCII table was and is such a popular standard with many programming languages, the Unicode designers decided to use the ASCII character set as a subset of the Unicode character set. They inserted the ASCII character set's characters in the first 128 slots of the Unicode character set. That means programmers can find those characters' numeric values by referring to a simple ASCII table; they don't have to wade through the enormous Unicode character set.

Unicode Escape Sequence

Whenever you write an integer, you can write it in either decimal format or hexadecimal format. Likewise, you can specify a character by writing its numeric value in either decimal format or hexadecimal format and then casting it with the char cast operator. Java also provides another way to specify a character. You can use the *Unicode escape sequence*. The Unicode escape sequence is \u followed immediately by the hexadecimal digits of a hexadecimal number. Here's what we're talking about:



```
'\u####'
```

Each # stands for one hexadecimal digit. We elected to show this in single quotes, not double quotes, to emphasize that the six-element escape sequence is just a single character, not a string. It's just like any other escape sequence, however, so you can embed the \u#### anywhere in a string. The u must be lowercase, and there must be exactly four hexadecimal digits.8

Using Unicode in Java Programs

If you want to print characters using Unicode escape sequences, you can use System.out.println in a text-based environment for the first 128 characters, but for the other characters, System.out.println in a text-based environment doesn't work consistently. That's because text-based environments recognize just the ASCII portion of the Unicode table; that is, the first 128 characters. To print all the characters in the Unicode table, you need to use graphical user interface (GUI) commands in a GUI environment.

The program in Figure 12.16 provides a GUI window and uses it to illustrate a small sampling of the many characters that are available. The start method's declarations section creates a codes array for int code values, which are the Unicode escape sequences for the first characters in blocks of characters we want to display. In the initializing process, these Unicode escape sequences automatically promote from type char to type int. The subsequent descriptions array is initialized with a simple String description of each block of characters.

After creating a particular font, the next declaration creates a single TextArea object named area and initializes it with information about the previously created font. Then we add that area object to a 600 × 400 pixels Scene. The first statement after the declarations gives the stage a title. The next two statements configure area for automatic text wrapping and no user editing.

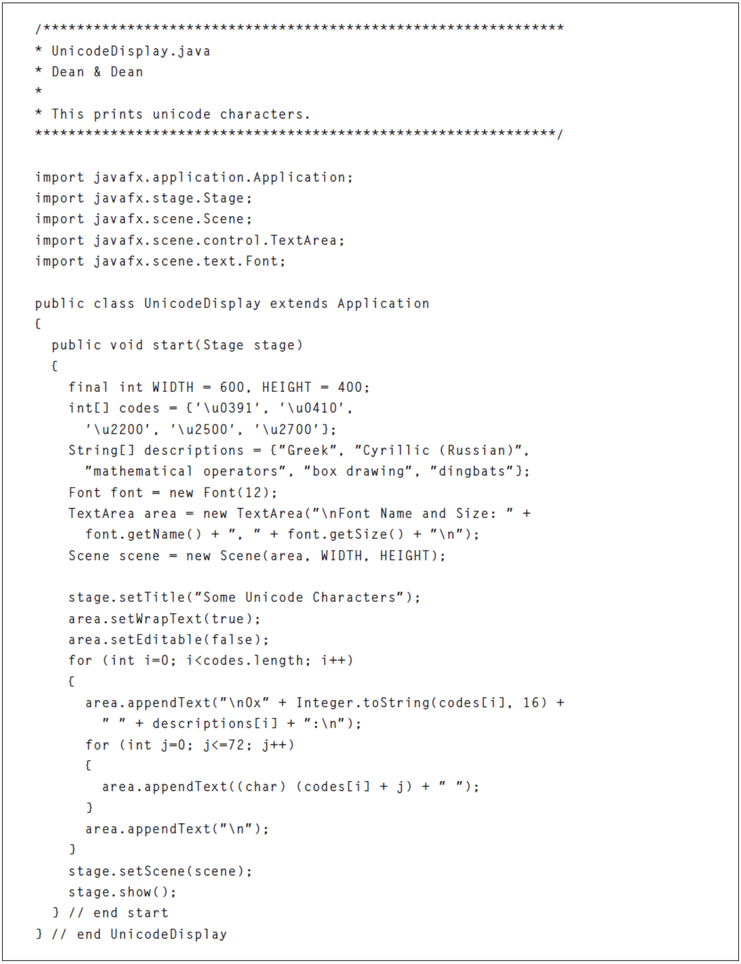
Then an outer loop iterates over the

five blocks of Unicode. At the beginning of each iteration, the area object calls its appendText method to add a description of that particular block of Unicode. Then in a nested for loop, area calls appendText again to add each of the first 73 characters in that block. In the appendText method's argument, notice how we add the nested loop's index variable, j, to the initial Unicode value to identify each individual Unicode value as an int. Then we cast that int into a char. Finally, we concatenate " " to the character, which produces a string, which matches the appendText method's parameter type.

Figure 12.17 shows the GUI output this program generates. The characters in the codes array in Figure 12.16 are the Unicode escape sequences for the first character in each block of characters shown in Figure 12.17. Notice that both the Greek and Cyrillic blocks include both uppercase and lowercase characters, and they include some additional characters beyond the normal final values of Ω (ω) and **Я** (я), respectively. These (and other) additional characters are needed for some of the individual languages in the families of languages using these alphabets. Of course, the characters shown in Figure 12.17 are just a tiny sampling of all the characters in Unicode.

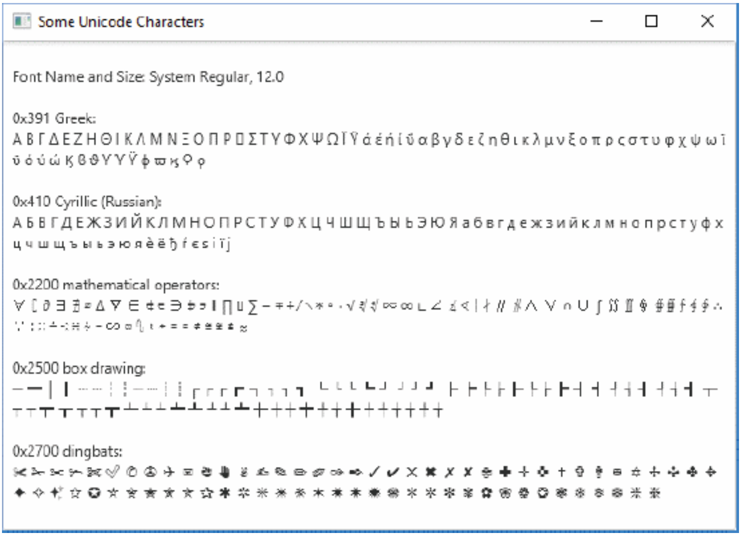
Notice that the different characters shown in Figure 12.17 have generally different widths. To get constant-width characters, you'd have to change the font type to something like Courier New. You could do that—and also change the style to bold and size to 10 points—by inserting a statement like this:

```
area.setFont(new Font("Courier New", Font.BOLD, 10));
```



```
/*******************************************************************
* UnicodeDisplay.java
* Dean & Dean
*
* This prints unicode characters.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.TextArea;
import javafx.scene.text.Font;

public class UnicodeDisplay extends Application
{
  public void start(Stage stage)
  {
    final int WIDTH = 600, HEIGHT = 400;
    int[] codes = {'\u0391', '\u0410',
      '\u2200', '\u2500', '\u2700'};
    String[] descriptions = {"Greek", "Cyrillic (Russian)",
      "mathematical operators", "box drawing", "dingbats"};
    Font font = new Font(12);
    TextArea area = new TextArea("\nFont Name and Size: " +
      font.getName() + ", " + font.getSize() + "\n");
    Scene scene = new Scene(area, WIDTH, HEIGHT);

    stage.setTitle("Some Unicode Characters");
    area.setWrapText(true);
    area.setEditable(false);
    for (int i=0; i<codes.length; i++)
    {
      area.appendText("\n0x" + Integer.toString(codes[i], 16) +
        " " + descriptions[i] + ":\n");
      for (int j=0; j<=72; j++)
      {
        area.appendText((char) (codes[i] + j) + " ");
      }
      area.appendText("\n");
    }
    stage.setScene(scene);
    stage.show();
  } // end start
} // end UnicodeDisplay
```

Figure 12.16 Program that uses GUI to display a sampling of Unicode characters



```
Font Name and Size: System Regular, 12.0

0x391 Greek:
Α Β Γ Δ Ε Ζ Η Θ Ι Κ Λ Μ Ν Ξ Ο Π Ρ ΢ Σ Τ Υ Φ Χ Ψ Ω Ϊ Ϋ ά έ ή ί ΰ α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ ς σ τ υ φ χ ψ ω ϊ
ϋ ό ύ ώ Ϗ ϐ ϑ ϒ ϓ ϔ ϕ ϖ ϗ Ϙ ϙ

0x410 Cyrillic (Russian):
А Б В Г Д Е Ж З И Й К Л М Н О П Р С Т У Ф Х Ц Ч Ш Щ Ъ Ы Ь Э Ю Я а б в г д е ж з и й к л м н о п р с т у ф х
ц ч ш щ ъ ы ь э ю я ѐ ё ђ ѓ є ѕ і ї ј

0x2200 mathematical operators:
∀ ∁ ∂ ∃ ∄ ∅ ∆ ∇ ∈ ∉ ∊ ∋ ∌ ∍ ∎ ∏ ∐ ∑ − ∓ ∔ ∕ ∖ ∗ ∘ ∙ √ ∛ ∜ ∝ ∞ ∟ ∠ ∡ ∢ ∣ ∤ ∥ ∦ ∧ ∨ ∩ ∪ ∫ ∬ ∭ ∮ ∯ ∰ ∱ ∲ ∳ ∴
∵ ∶ ∷ ∸ ∹ ∺ ∻ ∼ ∽ ∾ ∿ ≀ ≁ ≂ ≃ ≄ ≅ ≆ ≇ ≈

0x2500 box drawing:
─ ━ │ ┃ ┄ ┅ ┆ ┇ ┈ ┉ ┊ ┋ ┌ ┍ ┎ ┏ ┐ ┑ ┒ ┓ └ ┕ ┖ ┗ ┘ ┙ ┚ ┛ ├ ┝ ┞ ┟ ┠ ┡ ┢ ┣ ┤ ┥ ┦ ┧ ┨ ┩ ┪ ┫ ┬
┭┮┯┰┱┲┳┴┵┶┷┸┹┺┻┼┽┾┿╀╁╂╃╄╅╆╇╈

0x2700 dingbats:
✀ ✁ ✂ ✃ ✄ ✅ ✆ ✇ ✈ ✉ ✊ ✋ ✌ ✍ ✎ ✏ ✐ ✑ ✒ ✓ ✔ ✕ ✖ ✗ ✘ ✙ ✚ ✛ ✜ ✝ ✞ ✟ ✠ ✡ ✢ ✣
✤ ✥ ✦ ✧ ✨ ✩ ✪ ✫ ✬ ✭ ✮ ✯ ✰ ✱ ✲ ✳ ✴ ✵ ✶ ✷ ✸ ✹ ✺ ✻ ✼ ✽ ✾ ✿ ❀ ❁ ❂ ❃ ❄ ❅ ❆ ❇ ❈
```

Figure 12.17 Output produced by the program in Figure 12.16

©Oracle/Java

Suppose you want the Unicode value for ≈. That's the last mathematical operator displayed in Figure 12.17. As indicated by the third codes value in the UnicodeDisplay program, the first mathematical operator has a Unicode hexadecimal value of 0x2200. The maximum value of the inner for loop in Figure 12.16 is 72. The hexadecimal value of 72 is 4 × 16 + 8 = 0x0048. Thus, the Unicode hexadecimal value of the last mathematical operator displayed in Figure 12.17 is 0x2200 + 0x0048 = 0x2248. Sometimes you can use a word processor to find the Unicode value of a symbol you're interested in. For example, in Microsoft Word, select Insert / Symbol / Mathematical Operators, and then select ≈. Then read the Unicode hex value for the selected symbol from the "Character code" field near the bottom of the Symbol window. This shows that the Unicode hexadecimal value for the ≈ character is 0x2248.

By browsing through the https://unicode.org website, you can find everything Unicode has to offer. If you go there, look for a Code Charts link, and click on it. That should take you to a page that lets you explore the various subtables within the huge Unicode table. Try to find the basic Latin link. That takes you to the Basic Latin subtable, which is equivalent to the

ASCII table that we present in -Appendix 1. This particular Unicode subtable is called Latin because it contains the Latin alphabet—a, b, c, and so on. Visit a few of the other subtables to get an idea of what's available. In every subtable, you'll see a set of characters, and for each character, you'll see its equivalent Unicode value.

There are also several other standards for assigning numbers to characters. Computer applications sometimes include translation tables to make conversions between their own character-coding schemes and Unicode. Be warned, however. The translations don't always work as you might like, and special characters may change in surprising ways when you transfer text with special characters from one application to another.

12.17 Introduction to

GridWorld Case Study (Optional)

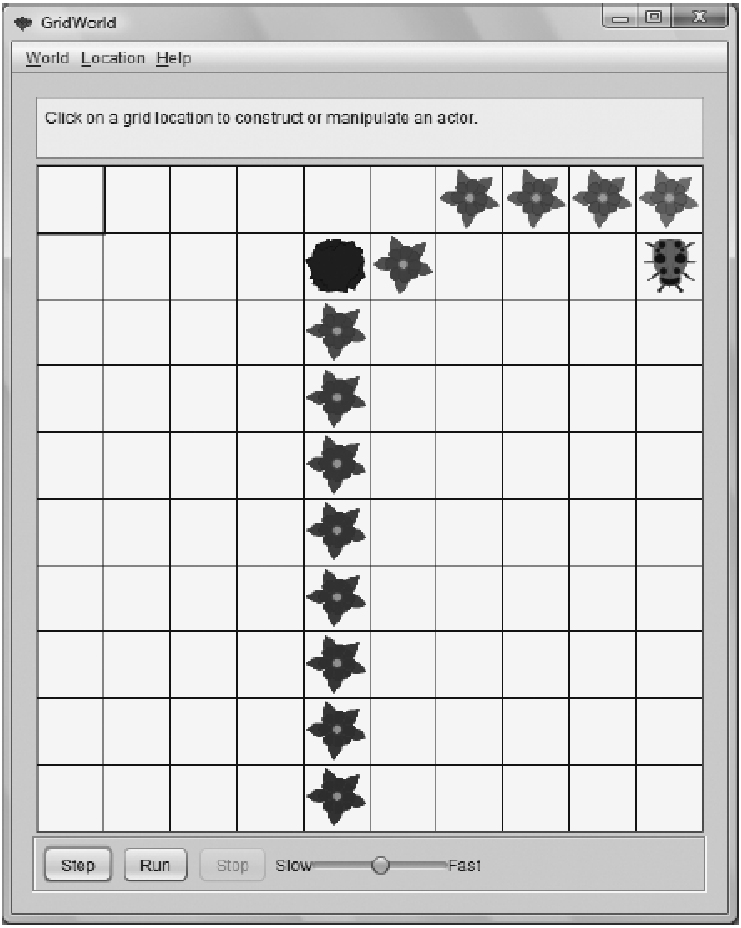
This optional section introduces an extended example referred to as GridWorld,[9] which shows how different Java classes work together in the context of a larger program. Because the classes in this case study are distributed among several different packages, you may want to look at Appendix 4 before proceeding further.

GridWorld is a GUI program, but it was written before the introduction of JavaFX. So, it provides a relatively large example of legacy Java GUI. Because our extensions to GridWorld also rely on legacy Java GUI, you can think of this section and the accompanying one at the end of the next chapter and several exercises at the ends of this and the next two chapters as optional GUI material that illustrates use of legacy Java GUI.

Go to https://apstudent.collegeboard.org/apcourse/ap-computer-science-a/about-the-exam/gridworld-case-study/ to install GridWorld software and documentation on your computer. Then download and unpack the GridWorldCode.zip file. This file contains a framework directory (the source code for the core classes used by GridWorld programs), a javadoc directory (documentation described in Section 8.3 of Chapter 8 and Appendix 6), and a projects directory (source code for particular GridWorld program drivers). The GridWorldCode.zipfile also contains a gridworld.jar file, which has bytecode for the core classes used by GridWorld programs. This bytecode is in four packages: info.gridworld.actor, info.gridworld.grid, info.gridworld.GUI, and info.gridworld.World. To import these packages, your compiler must know where they are. You can tell it where they are by including the path to the gridworld.jar file as an option each time that you call the javac compiler. (See the Installation Guide at the above website for details.)

GridWorld's "grid" is a two-dimensional "checkerboard" that holds pieces called "actors." Once you have installed the GridWorld software, try it out. Do this by going to the projects directory and then to the firstProject directory, where you should find a file called BugRunner. This file is the driver for an introductory demonstration. Compile BugRunner and then execute it. This should generate a GUI display with one randomly located rock and one randomly located bug, with the bug pointing up. Clicking a Step button in the lower-left corner of the display makes the bug move forward. As the bug moves, it deposits flowers in its wake, and as time passes, these flowers darken. Figure 12.18 shows a typical example, 16 steps after a start with the bug at the bottom of the screen. Whenever the bug encounters an obstacle like a rock or a boundary, instead of moving forward, it turns right by 45°, and it takes two steps to turn 90°.

This program imports code that defines three types of actors—rocks, bugs, and flowers. Program initialization automatically creates and randomly locates one black rock and one red bug. Clicking the Run button makes the bug run forward in repeated steps. You can control how fast the bug runs by adjusting the slow-fast slider. Once you click the Run button, the subdued Stop button brightens. Then you can click that Stop button to make the bug stop running.


GridWorld
World Location Help
Step
Run
Stop
Slow
Fast

Click on a grid location to construct or manipulate an actor.

Figure 12.18 Display for GridWorld's firstProject, produced by executing BugRunner and clicking Step 16 times Starting points are random, so different executions give different results.

©Oracle/Java

Clicking the little icon in the extreme-upper-left corner enables you to move, resize, maximize, minimize, or close the window. Clicking World enables you to alter the grid's dimensions or to quit. By default, the program initially selects the location in the upper-left corner and gives it a darker boundary. Clicking Location enables you to move the selected location, to edit, enter, or delete an actor at the selected location, or to zoom in or zoom out. An option under Help describes the results of various mouse actions and corresponding keyboard shortcuts. Within the grid, clicking an empty location displays a list of constructors. Clicking one of those constructors inserts a rock, bug, or flower into that square. Clicking an occupied location displays a list of the methods you can invoke for the actor currently at that location.

In the next two chapters, additional

reading and additional exercises will help you investigate existing program structure and behavior and modify program code to alter the behavior.

Summary

- Numerical overflow creates dramatic errors. Whenever there is any doubt about a particular type’s ability to hold a value that might be assigned to it, change to a larger type.
- Floating-point numbers have a greater range than integers, but for a given amount of memory, they provide less precision.
- The ASCII character set provides numerical values for the symbols on a standard keyboard.
- Because characters are

represented as numbers, ch1 + ch2 evaluates to the sum of the ASCII values for the char variables, ch1 and ch2.

- Type casting allows you to put a numeric value into a numeric variable of a different type, but be careful that you don't get overflow or undesired truncation when you do it.

- When used as a prefix, an increment (++) or decrement (--) operator changes the variable value before that variable participates in other expression operations. When used as a postfix, an increment or decrement operator changes the variable value after it participates in other expression operations.

- If a statement contains multiple assignment operators, the rightmost assignment evaluates first.

- It’s sometimes helpful to embed an assignment within a condition, but you should avoid excessive use of embedded increment, decrement, and assignment operations.
- A conditional operator expression provides a compact conditional evaluation. If what’s before the ? is true, use what’s after the ?. Otherwise, use what’s after the:.
- Short-circuit evaluation means that the JVM stops evaluating an expression whenever the expression’s outcome becomes certain. Use this feature to avoid illegal operations.
- Use a break statement sparingly to terminate loops prematurely.
- An enum is a fixed set of distinct

constants or a fixed set of objects having distinct identifiers and final instance values. It identifies a finite set of alternatives.

- The -> symbol denotes a lambda expression, which implements a functional interface and executes it immediately by supplying arguments on the left side of the arrow to an anonymous method defined on the right of the arrow.

- The :: symbol denotes a method reference to an existing Java API or programmer-defined method that implements a functional interface.

- A stream is a sequence of elements supporting sequential and parallel sourcing, intermediate, and terminal operations.

- The original Unicode Standard provides numerical codes for more than 65,000 different characters. You can specify them as decimal or hexadecimal integers or with a Unicode escape sequence. To see the Unicode characters for codes above 127, you must display them in a GUI window.

Review Questions

---

§12.2 Integer Types and Floating-Point Types

1. For each integer data type, how many bits of storage are used?

2. How would you write the decimal constant, 1.602×10^{-19}, as a double?

3. What is the approximate precision (number of accurate decimal digits) of each of the floating-point types?

§12.3 char Type and the ASCII Character Set

4. How many distinct characters are identified by the basic ASCII character set?

5. What number can you add to an uppercase-letter char variable to convert it to lowercase?

§12.4 Type Conversions

Page 582

6. Assume the declaration:

public final double C = 3.0E10; // speed of light in cm/sec

Write a Java print statement that uses a cast operator to display the value of C in this format:

30000000000

7. Will this statement be OK, or will it generate a compile-time error? (OK / error)

```
float price = 66;
```

8. Will this statement be OK, or will it generate a compile-time error? (OK / error)

```
boolean done = (boolean) 0;
```

9. Will this statement be OK, or will it generate a compile-time error? (OK / error)

```
float price = 98.1;
```

§12.5 Prefix/Postfix Modes for Increment/Decrement Operators

10. What is the value of z after these statements execute?

```
int z, x = 3;

z = --x;

z + = x--;
```

§12.6 Embedded Assignments

11. Write one Java statement that makes w, x, and y all equal to the current value of z.

§12.7 Conditional Operator Expressions

12. Suppose x equals 0.43. Given the following switch construct heading, what does the controlling expression evaluate to?

switch (x>0.67 ? 'H' : (x>0.33 ? 'M' : 'L'))

§12.8 Expression Evaluation Review

13. Assume this:

int a = 2;

int b = 6;

float x = 8.0f;

Evaluate each of the following expressions, using these guidelines:

- As shown in Section 12.8, put each evaluation step on a separate line and use the ⇒ symbol between steps.

- Evaluate each expression independently of the other expressions; in other words, use the above assumed values for each expression evaluation.
- Expression evaluation problems can be tricky. We encourage you to check your work by running test code on a computer.
- If there would be a compilation error, specify “compilation error.”

a) a + 25 / (x + 2)
b) 7 + a * --b / 2
c) a * --b / 6
d) a + b++
e) a - (b = 4) % 7
f) b = x = 23

§12.9 Short-Circuit Evaluation

14. Assume expr1 and expr2 are expressions that evaluate to boolean values. Assume that expr1 evaluates to true. When the computer evaluates each of the following expressions, will it evaluate expr2? If yes, just say "yes." If no, explain why, and use the term "short-circuit evaluation" in your explanation.

- **a)** expr1 || expr2
- **b)** expr1 && expr2

15. Assume this:

```
int a = 2;
boolean flag = true;
```

Evaluate the following expression:

```
a < 3 || flag && !flag
```

§12.10 Empty Statement

16. Assume that the following code fragment is inside of a program that compiles successfully. What does the code fragment print? Hint: This is a trick question. Study the code carefully.

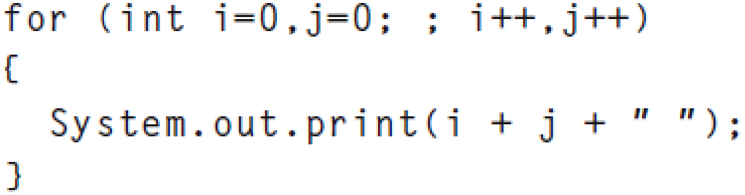
```
int x = 1;
while (x < 4);
{
  System.out.println(x);
  x++;
}
```

§12.11 Using break to Exit from a Loop

17. Usually, you should avoid using break statements in loops because using break statements forces readers to look for termination conditions inside loop bodies. (T / F)

§12.12 for Loop Header Details

18. Assume that the following code fragment is inside of a program that compiles successfully. What does the code fragment print?



```
for (int i=0,j=0; ; i++,j++)
{
  System.out.print(i + j + " ");
}
```

§12.13 Enumerated Types

19. Declare a set of private constants called “Kingdom” that identify the five kingdoms of life: Monera, Protoctista, Fungi, Plantae, and Animalia.

§12.16 GUI Track: Unicode (Optional)

20. What is the hexadecimal symbol for the decimal number 13?

21. The Unicode values for characters are the same as the ASCII values in the range 0x00 to 0xFF. (T / F)

Exercises

1. [after §12.2] If a double value overflows, what happens? What happens if a program divides by the result? Is an error produced, and if so what type of error?

2. [after §12.2] The following program illustrates using BigInteger and BigDecimal. Replace the <*insert code here*> lines with two assignment statements. As indicated by the output, the first assignment statement divides the original number by 3.0 and uses 30 significant digits in displaying the result. The second assignment statement divides the original number by 3.0 and uses 4 fractional digits in displaying the result.

```
import java.math.*; // Double, BigDecimal, MathContext;

public class Big
{
  public static void main(String[] args)
  {
    BigInteger bigI = new BigInteger("24242424242424242424242425");
    BigDecimal bigD = new BigDecimal(bigI);
    BigDecimal divisor = new BigDecimal(3.0);
    BigDecimal quotient;

    System.out.println("Original number: " + bigI);

    <insert code here>

    System.out.println("When divided by 3 and the result uses 30" +
      " significant digits:\n" + quotient);

    <insert code here>

    System.out.println("When divided by 3 and the result uses 4" +
      " fractional digits:\n" + quotient);  } // end main
} // end Big
```

Output:

```
Original number: 24242424242424242424242425
When divided by 3 and the result uses 30 significant digits:
8080808080808080808080808.33333
When divided by 3 and the result uses 4 fractional digits:
8080808080808080808080808.3333
```

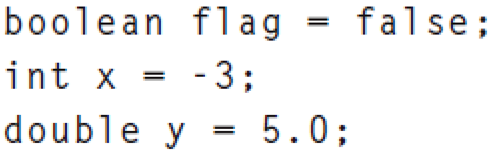
3. [after §12.3] Provide a robust method named getOddDigit which prompts the user to enter an odd digit and returns the odd digit as an int. Read the input as a string and verify that the input consists of a single odd digit. If the user enters invalid input, repeat the prompt and input.

4. [after §12.4] Provide a program that reads a line of text, retrieves the first character from the input line, and displays the subsequent character in the ASCII table.

5. [after §12.6] Assume c and d are boolean variables initialized as false. What are their values after this statement executes?

```
c = !((c=!(7< =6)) && (d=7> =6));
```

6. [after §12.7] Assume this:



```
boolean flag = false;
int x = -3;
double y = 5.0;
```

What is the value of the result after this statement executes?

```
int result = flag ? x : (int) (y - 0.7);
```

7. [after §12.8] Assume this:

```
int a = 4;

int b = 13;

double x = -9.1;
```

Evaluate each of the following

expressions. Follow these guidelines:

- As shown in Section 12.8, put each evaluation step on a separate line and use the ⇒ symbol between steps.

- Evaluate each expression independently of the other expressions; in other words, use the above assumed values for each expression evaluation.

- Expression evaluation problems can be tricky. We encourage you to check your work by running test code on a computer.

- If there would be a compilation error, specify "compilation error."

a) `a + b--`

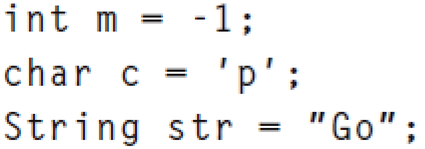
b) `a = x = -12`

c) 8 + a * + +b / 20

d) a + 7 / (x + 12.1)

e) a + (b = 5) % 9

8. [after §12.8] Assume this:



```
int m = -1;
char c = 'p';
String str = "Go";
```

Evaluate each of the following expressions. Follow these guidelines:

- As shown in Section 12.8, put each evaluation step on a separate line and use the ⇒ symbol between steps.

- Evaluate each expression independently of the other expressions; in other words, use the above assumed values for each expression evaluation.

- Expression evaluation problems can be tricky. We encourage you to check your work by running test code on a computer.

- If there would be a compilation error, specify “compilation error.”

  a) str + m + 3

  b) str + (m + 3)

  c) c + m

  d) 7 + '2'

  e) "\"" + str + '!'

9. [after §12.9] Consider the following code fragment. Line numbers are at the left.

```
1   int x = 5;
2   boolean y = true;
3   boolean z;
4   z = y && ++x == 6;
5   y = x++ == 3;
6   y = !z;
7   System.out.println(x + " " + y + " " + z);
```

Trace the code using this trace setup:

line#	x	y	z	output

Does short-circuit evaluation occur in line 4? Why or why not?

10. [after §12.9] Assume:

```
boolean x;
boolean y = false;
double z = 9.5;
```

Determine the output of this code fragment:

```
x = y || (--z == 8.5);
y = !x && (--z == 7.5);
System.out.println(z + " " + y + " " + x);
```

11. [after §12.10] What happens if you try to compile the following code, and why does it happen?

```
public static void main(String[] args)
{
  int number = 4;
  int factorial = 1;

  for (int i=1; i<=number; i++);
  {
    factorial *= i;
  }
  System.out.println("factorial(" + number + ") = " + factorial);
}
```

12. [after §12.12] The Fibonacci sequence is a sequence of numbers such that the first two numbers are 0 and 1, and each successive number is the sum of the two previous numbers. Thus, here are the first nine numbers in the Fibonacci sequence:

0 1 1 2 3 5 8 13 21

> The third number is 1 because the previous two numbers, 0 and 1, add up to 1. The fourth number is 2 because the previous two numbers, 1 and 1, add up to 2.

Provide a code fragment that prints the Fibonacci sequence numbers for the numbers less than 25 (i.e., the numbers shown above). Your code fragment must consist of just a for loop header terminated by a semicolon—nothing else. Hint: Use comma-separated initialization and update components. Feel free to use the following as a start for your for loop:

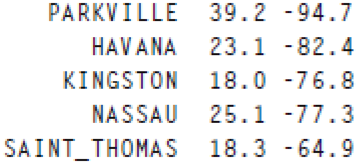
```
for (int m = 0, n = 1, temp;
```

> By the way, we recommend that you avoid using code like this for your real programs. This exercise is just for fun (fun for a hacker, anyway ☺).

13. [after §12.12] The following program works, but it's ugly. Modify it to perform the same function with a more conventional while loop.

```
import java.util.Scanner;

public class Test
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String allEntries="";

    for (String entry="";;)
    {
      System.out.print("Make entry or 'q' to quit: ");
      entry = stdIn.nextLine();
      if (entry.equals("q")) break;
      else allEntries += "  " + entry;
    } // end for
    System.out.println(allEntries);
  } // end main
} // end class Test
```

14. [after §12.13] As you learned earlier, an enumerated type is a class (albeit a special kind of class). As such, you can add a main method to it and have that main method run the enumerated type as a program. Enhance this chapter's City enumerated type by adding a main method that uses a for-each loop with a printf statement to display all cities and their properties. Specifically, it should display each city name followed by its latitude and longitude, like this:



```
  PARKVILLE  39.2 -94.7
     HAVANA  23.1 -82.4
   KINGSTON  18.0 -76.8
     NASSAU  25.1 -77.3
SAINT_THOMAS  18.3 -64.9
```

15. [after §12.13] Improve the CityTravel program by prompting the user to enter origin and destination cities and using the user-entered values in the getDistance method call. If the user does not spell one of the enumerated cities correctly, print an error message and loop until one of the enumerated cities is entered correctly. Do this by providing a new method with this signature:

public static City getCity(String

originOrDestination)

And in the modified main method, replace:

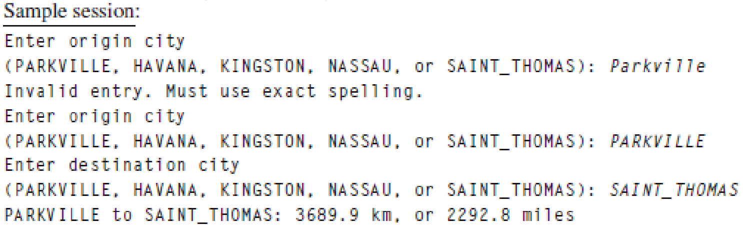
City origin = City.PARKVILLE;

City destination = City.KINGSTON;

with this:

City origin = getCity("origin");

City destination = getCity("destination");



Sample session:

```
Enter origin city
(PARKVILLE, HAVANA, KINGSTON, NASSAU, or SAINT_THOMAS): Parkville
Invalid entry. Must use exact spelling.
Enter origin city
(PARKVILLE, HAVANA, KINGSTON, NASSAU, or SAINT_THOMAS): PARKVILLE
Enter destination city
(PARKVILLE, HAVANA, KINGSTON, NASSAU, or SAINT_THOMAS): SAINT_THOMAS
PARKVILLE to SAINT_THOMAS: 3689.9 km, or 2292.8 miles
```

16. [after §12.14] Modify the StockAverage2 program in Figure 12.13 by inserting into the stream’s method-call chain a .peek method call that prints the previously entered values. Use a lambda expression for peek’s argument.

17. [after §12.16] What is the Unicode hexadecimal value for Σ (uppercase Greek sigma)? Show or explain how you got your answer.

18. [after §12.17] (Case Study) Run GridWorld’s BugRunner program. Perform the following operations and answer the questions:

a) Move the rock to the location of the bug. Then move the rock to location (6, 0).

What happens when you move a rock to the location of a bug?

b) Insert a second rock at location (3, 9). Insert a red bug at (4, 4). Insert a blue bug at (5, 5). Adjust the directions of the two bugs so that they face each other.

After they face each other, what is the red bug's direction and what is the blue bug's direction?

c) Click "Step" and see what happens. Then click "Step" again and see what happens. Then Click "Run." Adjust the speed and watch the behavior until you see regular repetition. Then stop the bug from running and delete one of the bugs.

What rules govern bug direction and motion?

What rules govern flower creation, deletion, and coloration?

Review Question

Solutions

1. byte = 8 bits, short = 16 bits, int = 32 bits, long = 64 bits

2. 1.602E-19 or 1.602e-19

3. float precision ≈ 6 digits, double precision ≈ 15 digits

4. The basic ASCII character set describes 128 different characters.

5. To convert uppercase to lowercase, add 32. To go the other way, subtract 32.

6. System.out.println((long) C); // (int) isn't big enough!

7. This statement is OK:

```
float price = 66;
```

8. This statement generates a compile-time error because it’s illegal to convert between numeric values and boolean values:

```
boolean done = (boolean) 0;
```

9. This statement generates a compile-time error because floating-point constants are double by default:

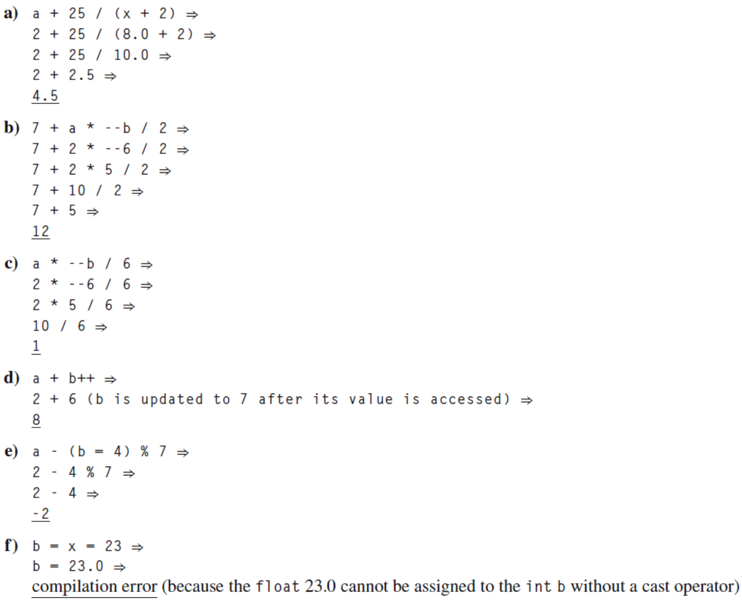
float price = 98.1;

10. z's value is 4. The first decrement uses prefix mode so x is first decremented to 2, then 2 is assigned into z. The second decrement uses postfix mode so x is decremented after its value of 2 is added to z.

11. w = x = y = z; or any other sequence that has z on the right.

12. The switch controlling expression evaluates to 'M'

13. Expression-evaluation practice:



a)
```
a + 25 / (x + 2) ⇒
2 + 25 / (8.0 + 2) ⇒
2 + 25 / 10.0 ⇒
2 + 2.5 ⇒
4.5
```

b)
```
7 + a * --b / 2 ⇒
7 + 2 * --6 / 2 ⇒
7 + 2 * 5 / 2 ⇒
7 + 10 / 2 ⇒
7 + 5 ⇒
12
```

c)
```
a * --b / 6 ⇒
2 * --6 / 6 ⇒
2 * 5 / 6 ⇒
10 / 6 ⇒
1
```

d)
```
a + b++ ⇒
2 + 6 (b is updated to 7 after its value is accessed) ⇒
8
```

e)
```
a - (b = 4) % 7 ⇒
2 - 4 % 7 ⇒
2 - 4 ⇒
-2
```

f)
```
b = x = 23 ⇒
b = 23.0 ⇒
```
compilation error (because the `float` 23.0 cannot be assigned to the `int` b without a cast operator)

Page 589

14. Will it evaluate expr2

a) No. Since the left side of the || operator is true, short-circuit evaluation will cause the right side of the || operator (expr2) to be ignored (since the result of the entire expression will evaluate to true regardless of expr2's value).

b) Yes.

15. Assuming:

```
int a = 2;
boolean flag = true;

a < 3 || flag && !flag ⇒
2 < 3 || true && !true ⇒
2 < 3 || true && false ⇒
true || true && false ⇒
true      (short-circuit evaluation dictates "true or anything" evaluates to true)
```

16. It prints nothing because, due to the empty statement, the while loop header executes repeatedly in an infinite loop.

17. True. Normally, you should avoid using break statements in loops.

Page 590

18. The code fragment generates an infinite loop because the for loop header's missing second component is true by default. The output is:

0 2 4 6 ...

19. private enum Kingdom

```
{
  MONERA, PROTOCTISTA, FUNGI, PLANTAE, ANIMALIA;
}
```

20. The hexadecimal symbol for the decimal number 13 is either d or D.

21. False. They are the same only in the range from 0x00 to 0x7F.

---

[1] See Chapter 1's "First Program—Hello World" section for a description of how to run a program in a console window.

[2] The compound statement, defined in Chapter 4, is a group of zero or more statements surrounded by braces.

[3] This adds a delay of 1,000 milliseconds (which equals 1 second):

[4] The compiler automatically creates a values method that matches your enumerated type definition.

[5] Streams typically lead to more compact code. However, there is often a concomitant penalty in execution time. To quantify this time penalty, we compared execution times on the same computer for all code after the while loop in the StockAverage and StockAverage2 programs. This part of the StockAverage program consumed

about 2.8 milliseconds, whereas the same part of the StockAverage2 program consumed about 7.9 milliseconds. Thus (on a 2017 Dell Inspiron computer) the shorter streaming calculation increased the execution time by a factor of three.

[6] "Collections Framework Overview," https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html.

[7] We're focusing on the original Unicode standard, which is a subset of the current Unicode standard. The original Unicode standard is good enough for almost all Java programming. The original Unicode standard uses 16 bits for all characters. The current Unicode standard uses additional bits for additional characters that can't fit in the original Unicode set of 65,536 values. For additional details, see https://

www.unicode.org/standard/standard.html.

[8] The supplementary Unicode characters have numeric values that require more than four hexadecimal digits. To specify one of these supplementary characters, use a decimal or hexadecimal int representation of the character, or prefix the \u-representation of the four least-significant hexadecimal digits with an appropriate \u-representation in the range, \uD800 through \uDFFF. The prefix, called a *surrogate*, has no independent character association. (For additional information, see documentation on Java's Character class and https://www.unicode.org/Public/UNIDATA/Blocks.txt.) There's also another surrogate scheme that -represents characters with an 8-bit base value and multiple 8-bit surrogates. This latter scheme is used in communications.

[9] GridWorld software was written by Cay Horstmann, “GNU General Public License,” for the College Board Advanced Placement (AP) Program, https://www.gnu.org/gnu/gnu.html. This program is free for anyone to use, and no one may prevent anyone else from using it, although people may charge for their particular form of delivery. Thus, this software is “free” as in “free speech,” but not necessarily free as in “free beer.”

Aggregation, Composition, and Inheritance

Objectives

- Understand how things are naturally organized in aggregations and compositions.
- Implement aggregation and

composition relationships within a program.

- Understand how inheritance can be used to refine an existing class.
- Implement an inheritance hierarchy within a program.
- Learn how to write constructors for derived classes.
- Learn how to override an inherited method.
- Learn how to prevent overriding.
- Learn how to use a class to represent an association.

Outline

13.1 Introduction 592

13.2 Composition and Aggregation 592

13.3 Inheritance Overview 599

13.4 Implementation of a Person/Employee/FullTime Hierarchy 603

13.5 Constructors in a Subclass 605

13.6 Method Overriding 606

13.7 Using the Person/ Employee/FullTime Hierarchy 609

13.8 The final Access Modifier 610

13.9 Using Inheritance with Aggregation and Composition 610

13.10 Design Practice with Card Game Example 613

13.11 GridWorld Case Study Extensions (Optional) 619

13.12 Problem Solving with Association Classes (Optional) 626

13.1 Introduction

Page 592

Prior to this chapter, the programs you've created have been relatively simple in terms of their object orientation, so you've been able to describe all the objects in a program with just a single class. But for more complex programs, you should consider implementing multiple classes, one for each different type of -object within a program. In this chapter, you'll do just that, and you'll focus on the different ways to organize classes in a multiple-class program. First, you'll learn how to organize classes that are parts of a larger containing class. When classes are related like that, where one class is the whole and the other classes are parts of the whole, the classes form an *aggregation*. Then you'll learn how to organize classes where one class, the *base class,* defines common features for a group of objects, and the other classes define specialized features for each of the different types of objects in the group. When classes are related like that, the classes form an *inheritance* hierarchy. It's called an inheritance hierarchy because the specialized classes inherit features from the base class.

In describing inheritance, we present various techniques for working with an inheritance hierarchy's classes. Specifically, we present *method overriding,* which allows you to redefine a method in a specialized class that's already been defined in the base class. We also present the final modifier, which allows you to prevent a specialized class from overriding a method defined in the base class.

As a follow-up to the initial presentation of aggregation and inheritance concepts, we describe how the two design strategies can work together. It’s sometimes difficult to decide which is the best strategy to use. To give you practice with those decisions, we guide you partway through a program design activity and develop the skeleton for what could be a sophisticated card game program.

As new concepts are introduced, end-of-chapter exercises relate them to the GridWorld case study. An optional section near the end of the chapter uses several of this chapter’s concepts together in a GridWorld extension. Another optional section shows you how to improve organization by creating an association class, which defines a set of characteristics that belong to a particular relationship between classes.

By showing you how to organize multiple classes, this chapter provides you with important tools necessary to tackle real-world problems. After all, most real-world programming projects are large and involve multiple types of objects. When you organize objects correctly, it makes programs easier to understand and maintain. And that's good for everyone!

13.2 Composition and Aggregation

There are two primary forms of aggregation. As described above, standard aggregation is when one class is the whole and other classes are parts of the whole. The other form of aggregation also defines one class as the whole and other classes as parts of the whole. But it has an additional constraint that says the whole class is the exclusive owner of the part classes. "Exclusive ownership" means that the part classes cannot be owned by another class while they are being owned by the whole class. This exclusive-ownership form of aggregation is called *composition.* With composition, the whole class is called the *composite,* the part classes are called *components,* and the composite contains the components. Composition is considered to be a strong form of aggregation because the composite-component connections are strong (due to each component having only one owner—the composite).

Composition and Aggregation in the Real World

The concept of composition was not created for computer programming; it’s frequently used for complex objects in the real world. Every living creature and most manufactured products are made up of parts. Often, each part is a subsystem that is itself made up of its own set of subparts. Together, the whole system forms a composition hierarchy.

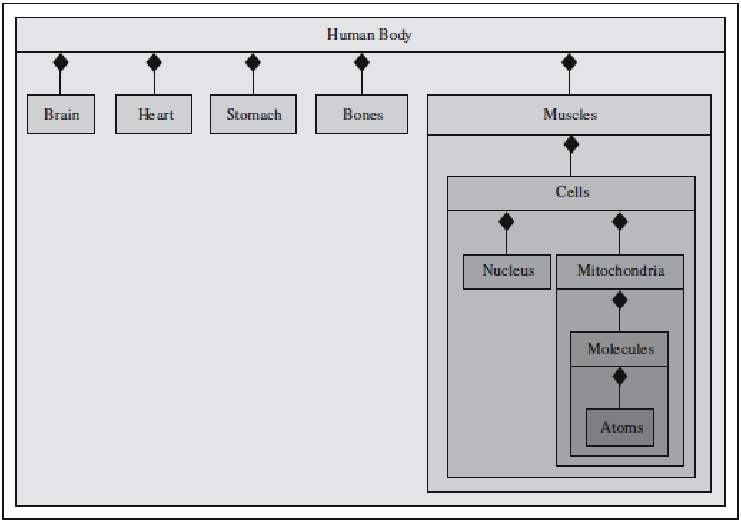

Human Body
Brain
Heart
Stomach
Bones
Muscles
Cells
Nucleus
Mitochondria
Molecules
Atoms

Figure 13.1 Partial representation of a composition hierarchy for a human body

Figure 13.1 shows a composition hierarchy for a human body. At the top of this particular composition hierarchy is a whole body. A human body is composed of several organs—brain, heart, stomach, bones, muscles, and so on. Each of these organs is in turn composed of many cells. Each of these cells is composed of many organelles, like the nucleus (a cell's “brain”), and the mitochondria, (a cell’s “stomach”). Each organelle is composed of many molecules. And finally, each organic molecule is composed of many atoms.

In a composition hierarchy (as well as in an aggregation hierarchy), the relationship between a containing class and one of its part classes is known as a *has-a* relationship. For example, each human body has a brain and has a heart. Remember that with a composition relationship, a component part is limited to just one owner at a time. For example, a heart can be in only one body at a time. Although the ownership is exclusive, it’s possible for the ownership to change. With a heart transplant, a heart can switch to a new owner, but it still has just one owner at a time.

Note the diamonds in Figure 13.1. In the Unified Modeling Language (UML), solid diamonds denote a composition relationship. They indicate that a whole has exclusive ownership of a part.

Now let’s think about an aggregation example where the parts are not exclusively owned by the whole. You can implement a school as an aggregation by creating a whole class for the school and part classes for the different types of people who work and study at the school. The people aren’t exclusively owned by the school because a person can be part of more than one aggregation. For example, a person can attend classes at two different schools and be part of two school aggregations. The same person might even be part of a third aggregation of a different type, like a household aggregation.

Composition and Aggregation in a Java Program

Let’s look at an example that uses both

class relationships—composition (where exclusive ownership is required) and standard aggregation (where exclusive ownership is not required). Suppose you're trying to model a car dealership with a computer program. Because the car dealership is made from several distinct, nontrivial parts, it's a good candidate for being implemented as an aggregation. The "whole" (the top of the aggregation hierarchy) is the dealership. Typically, a business has two kinds of "parts"—people and property. For simplicity, suppose that the only types of people at the car dealership are management and salespeople, and suppose the only type of property is cars. The control that the dealership has over the people is limited. They may also have other relationships, like memberships in families and social clubs. The dealership does not own its employees exclusively. Therefore, the relationship between the dealership and its employees is just aggregation. But the dealership does own its cars exclusively. So that relationship is composition. Note that the dealership can transfer ownership of its cars to its customers. That's OK because composition permits ownership to be transferred. Using a bottom-up design methodology, you should define three classes—Car, Manager, and SalesPerson—for the three types of component objects. Then, you should define a Dealership class for the container object.

Before you see the Dealership program's code, let's focus on the big-picture concepts using a UML class diagram. Figure 13.2's UML class diagram shows the Dealership program's four classes and the relationships among them, plus two other classes with which Manager or SalesPerson might be associated. Because we're now focusing on just the relationships among classes, in each representation of a class, we include just the class name and omit variables and methods. That's OK—UML is very flexible, and such omissions are allowed by the UML standards. UML indicates class relationships with connecting lines that run from one class to another. Formally, each connecting line is called an *association line.*

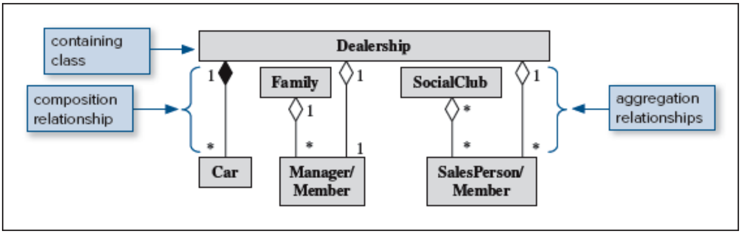
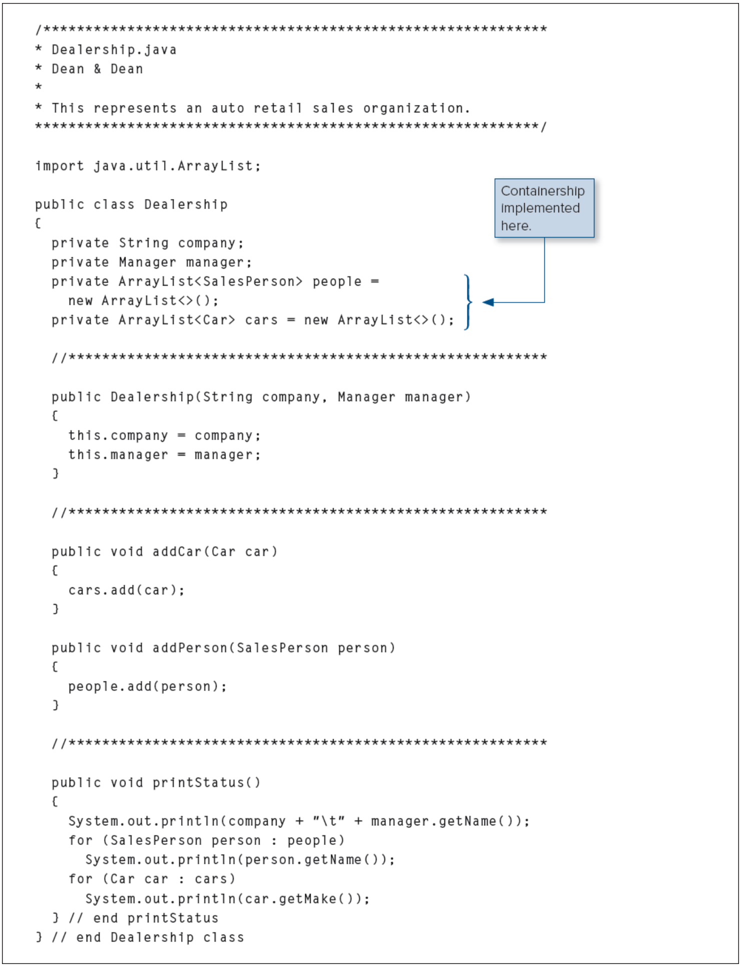

containing class
composition relationship
Dealership
Family
SocialClub
aggregation relationships
Car
Manager/ Member
SalesPerson/ Member
1
*
1
1
*
1
1
*
*
*
1
*

Figure 13.2 Class diagram for Dealership program

In Figure 13.2, note the diamonds on the association lines. Solid diamonds (like the one on the Dealership-Car line) indicate composition relationships, and hollow diamonds (like the ones on the Dealership-Manager and Dealership-SalesPerson lines) indicate aggregation relationships. The diamonds always go next to the containing class, so Figure 13.2's class diagram indicates that Dealership is the containing class.

Notice the numbers and asterisks written beside the association lines. These are *multiplicity values* that UML uses to specify the number of objects that participate in associations. The two 1's on the line between Dealership and Manager indicate a one-to-one association. That means there's one manager for each car dealership. If there were a 2 multiplicity value for the manager class, that would indicate two managers for each car dealership. The combination of 1 and * on other association lines indicates one- to-many associations, where "many" implies an indefinite number. That means you can have lots of cars (or none) and lots of salespeople (or none) for one car dealership.

It's now time to move from the conceptual phase, with emphasis on the dealership's UML class diagram, to the implementation phase, with emphasis on the Dealership program code. Note the Dealership class in Figure 13.3, and in particular, note the manager, people, and cars instance variables declared inside the Dealership class. Those instance variable declarations implement the concept of the dealership class containing the other three classes. The general rule is that whenever you have a class that contains another class, declare an instance variable inside the containing class such that the instance variable holds a reference to one or more of the contained class's objects.



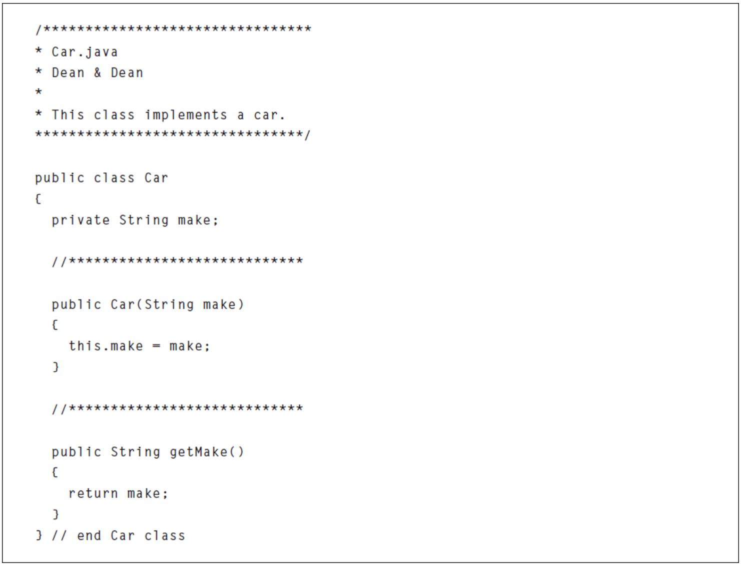
```
/************************************************************
* Dealership.java
* Dean & Dean
*
* This represents an auto retail sales organization.
************************************************************/

import java.util.ArrayList;

public class Dealership
{
  private String company;
  private Manager manager;
  private ArrayList<SalesPerson> people =
    new ArrayList<>();
  private ArrayList<Car> cars = new ArrayList<>();

  //*********************************************************

  public Dealership(String company, Manager manager)
  {
    this.company = company;
    this.manager = manager;
  }

  //*********************************************************

  public void addCar(Car car)
  {
    cars.add(car);
  }

  public void addPerson(SalesPerson person)
  {
    people.add(person);
  }

  //*********************************************************

  public void printStatus()
  {
    System.out.println(company + "\t" + manager.getName());
    for (SalesPerson person : people)
      System.out.println(person.getName());
    for (Car car : cars)
      System.out.println(car.getMake());
  } // end printStatus
} // end Dealership class
```

Containership implemented here.

Figure 13.3 Dealership class for the Dealership program

Also in the Dealership class, note the use of ArrayLists for the people and cars instance variables. Typically, if you have a class in a UML class diagram with a * multiplicity value, you can use an ArrayList to implement the reference to the asterisked class. ArrayLists are good for implementing * multiplicity values because they can expand to accommodate any number of elements.

Peruse the Car, Manager, and SalesPerson classes in Figures 13.4, 13.5, and 13.6. They simply store and retrieve data. Note the SalesPerson's sales instance variable—it keeps track of the total sales for a salesperson for the current year. There are no methods for accessing or updating the sales instance variable. We omitted those methods to avoid code clutter and to maintain focus on the matter at hand, aggregation and composition. In an actual car dealership program, you'd need to provide those methods.

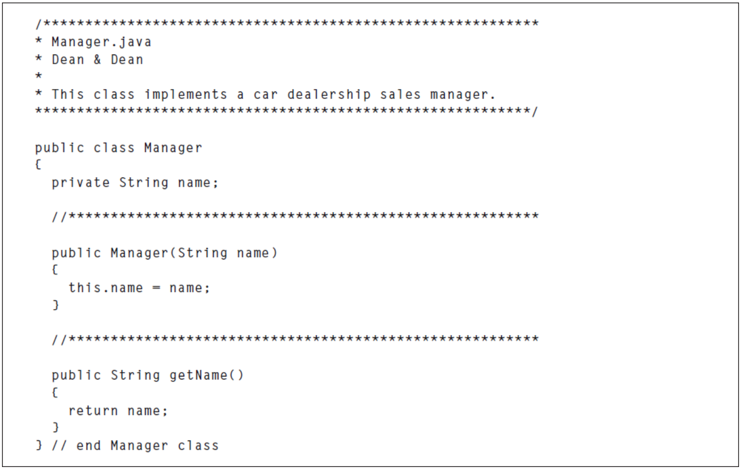


```
/********************************
* Car.java
* Dean & Dean
*
* This class implements a car.
********************************/

public class Car
{
  private String make;

  //***************************

  public Car(String make)
  {
    this.make = make;
  }

  //***************************

  public String getMake()
  {
    return make;
  }
} // end Car class
```

Figure 13.4 Car class for the Dealership program

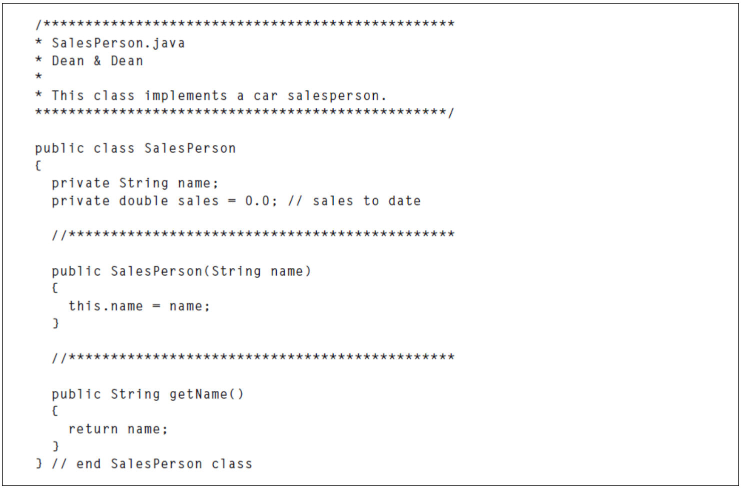
See the car dealership program's driver class in Figure 13.7. Most of the code is straightforward. The main method instantiates a Manager object, two SalesPerson objects, and a Dealership object. Then main adds salesPerson and Car objects to the Dealership object. The part of main that merits further attention is the use of local variables for the Manager and SalesPerson objects and the use of anonymous objects for the Car objects.

Why the discrepancy? Because Manager and SalesPerson relate to the Dealership class with aggregation, and Car relates to the Dealership class with composition.



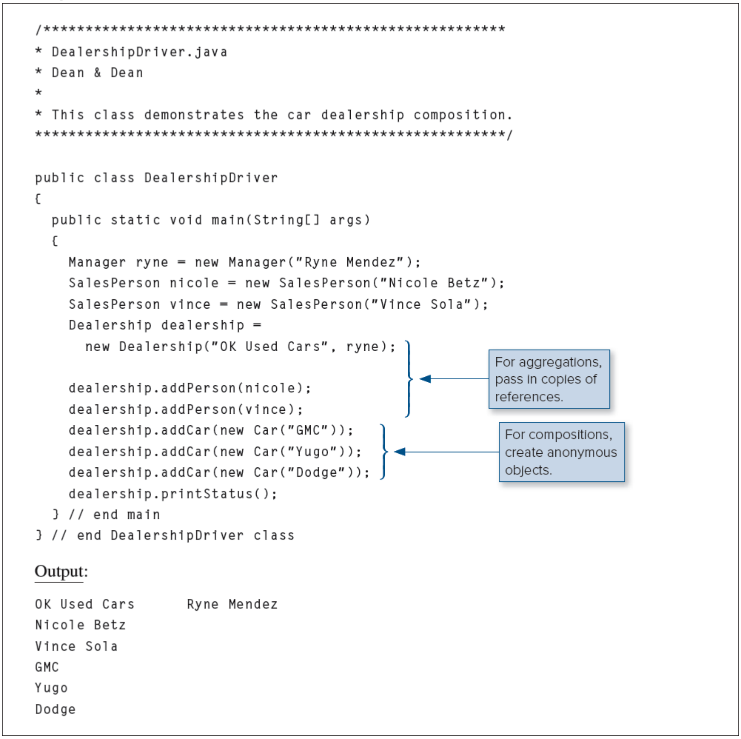
```
/*************************************************************
* Manager.java
* Dean & Dean
*
* This class implements a car dealership sales manager.
*************************************************************/

public class Manager
{
  private String name;

  //**********************************************************

  public Manager(String name)
  {
    this.name = name;
  }

  //**********************************************************

  public String getName()
  {
    return name;
  }
} // end Manager class
```

Figure 13.5 Manager class for the Dealership program



```
/*****************************************************
* SalesPerson.java
* Dean & Dean
*
* This class implements a car salesperson.
*****************************************************/

public class SalesPerson
{
  private String name;
  private double sales = 0.0; // sales to date

  //**********************************************

  public SalesPerson(String name)
  {
    this.name = name;
  }

  //**********************************************

  public String getName()
  {
    return name;
  }
} // end SalesPerson class
```

Figure 13.6 SalesPerson class for the Dealership program



```
/**********************************************************
* DealershipDriver.java
* Dean & Dean
*
* This class demonstrates the car dealership composition.
**********************************************************/

public class DealershipDriver
{
  public static void main(String[] args)
  {
    Manager ryne = new Manager(”Ryne Mendez”);
    SalesPerson nicole = new SalesPerson(”Nicole Betz”);
    SalesPerson vince = new SalesPerson(”Vince Sola”);
    Dealership dealership =
      new Dealership(”OK Used Cars”, ryne);

    dealership.addPerson(nicole);
    dealership.addPerson(vince);
    dealership.addCar(new Car(”GMC”));
    dealership.addCar(new Car(”Yugo”));
    dealership.addCar(new Car(”Dodge”));
    dealership.printStatus();
  } // end main
} // end DealershipDriver class
```

For aggregations, pass in copies of references.

For compositions, create anonymous objects.

Output:

```
OK Used Cars       Ryne Mendez
Nicole Betz
Vince Sola
GMC
Yugo
Dodge
```

Figure 13.7 Driver for the Dealership program

Here's the general rule for implementing aggregation relationships. Whenever two classes have an aggregation relationship, you should store the contained class's object in an instance variable in the containing class, and you should also store it in another variable outside the containing class. That way, the object can be added to another aggregation and have two different "owners" (having two different owners is allowed by aggregation). Putting this in the context of the Dealership program, DealershipDriver uses local variables when it instantiates the Manager and SalesPerson objects. That enables the Manager and SalesPerson objects to exist independently from the dealership, and that mirrors the real world.

Now let's look at the general rule for implementing composition relationships. Whenever two classes have a composition relationship, you should store the contained class's object in an instance variable in the containing class, and you should not store it elsewhere. That way, the object can have only one "owner" (having just one owner is required by composition). Putting this in the context of the Dealership program, DealershipDriver creates anonymous objects when it instantiates cars. That gives the dealership exclusive ownership and complete control over the cars, and that mirrors the real world.

13.3 Inheritance Overview

So far in this chapter, we've focused on aggregation and composition hierarchies, where one class is the whole and other classes are parts of the whole. Now we turn to classification hierarchies, which are qualitatively different from composition hierarchies. Whereas a composition hierarchy describes a nesting of things, a classification hierarchy describes an elaboration of concepts. The concept at the top is the most general/generic, and the concepts at the bottom are the most specific.

Classification Hierarchies and Inheritance of Attributes

Before looking at code, let's think about a real-world classification example. Figure 13.8 describes a few of the many possible characteristics of organisms living on the Earth today, with the most general characteristics at the top of the diagram and the most specific characteristics at the bottom of the diagram. Although this chart includes only characteristics of current living organisms, it's helpful to recognize that there was a natural time sequence in the development of these characteristics. The characteristics at the top developed first, and the characteristics at the bottom developed last. The earliest types of life—bacteria—appeared on Earth almost 4 billion years ago as single-celled organisms with no internal partitions. The "is alive" attribute listed in the box at the top—the most general class—endured and became an attribute common to all classes below it.

About 2.3 billion years ago, a nucleus and other components appeared inside cells, creating more sophisticated organisms called Eukaryotes. About 1.3 billion years ago, the first animals appeared. They had more than one cell, and they were vascular (had containers and conveyors like arteries and veins). About 510 million years ago, some of the animals (vertebrates) developed backbones and braincases. About 325 million years ago, the first reptiles appeared. Then, about 245 million years ago, the first mammals appeared. Thus, there is a correspondence between (1) the classification of biological attributes and (2) their sequence of development.

Start generic.

It's useful to identify a similar correspondence in object-oriented programming: (1) A classification hierarchy organizes a program's attributes. (2) A generic-to-specific development sequence organizes our implementation of those attributes. It's good design practice to start with a relatively simple and generic implementation and add specializations and complexity in subsequent design cycles. You'll see some examples of this later on.

With composition, certain classes contain other classes, but with inheritance, there’s no such containership. For example, in Figure 13.8, Animal is above Mammal, but an animal does not contain a mammal. Rather, animal is a generic type, and mammal is a specialized version of animal.

Each descendant type of organism inherits some attributes from its ancestors and adds some new attributes of its own. Ideally, the characteristics associated with each type high in the hierarchy should be just those attributes that are “conserved”—actually inherited by all types descended from that type. Thus, ideally, any type at the bottom of the hierarchy inherits all of the attributes associated with all types above it. For example, mammals have mammary glands and hair. And, because mammals are vertebrates, they inherit the vertebrate attributes of having a backbone and a braincase. And, because mammals are also animals, they also inherit the animal attributes of having more than one cell and being vascular. And, because mammals are also eukaryotes, they also inherit the eukaryote attribute of having a nucleus in each cell.

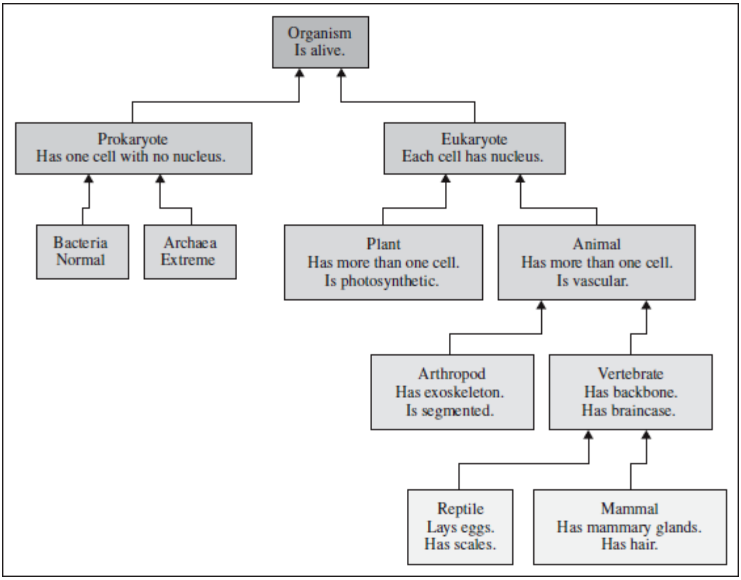

Organism
Is alive.
Prokaryote
Has one cell with no nucleus.
Eukaryote
Each cell has nucleus.
Bacteria
Normal
Archaea
Extreme
Plant
Has more than one cell.
Is photosynthetic.
Animal
Has more than one cell.
Is vascular.
Arthropod
Has exoskeleton.
Is segmented.
Vertebrate
Has backbone.
Has braincase.
Reptile
Lays eggs.
Has scales.
Mammal
Has mammary glands.
Has hair.

Figure 13.8 Biological example of a classification or inheritance hierarchy

Plan to Instantiate leaves only.

The types at the very bottom of a biological inheritance hierarchy do not appear in Figure 13.8, because the complete hierarchy is too big to display in one figure. What's actually at the bottom are species, like Homo sapiens (human beings). In nature, reproduction is possible only among members of the same species. Similarly, in an ideal OOP computer program, the only *realizable* (instantiable) types are the types at the very bottom of inheritance hierarchies. Ideally, all types above instantiable types are abstract types. In the next chapter, you'll learn how you can use Java's abstract keyword to prevent the instantiation of abstract types. Organizing an inheritance hierarchy so that all realizable (instantiable) types appear only at the lowest level (the leaves of a hierarchical tree) minimizes duplication, and it minimizes maintenance and enhancement labor.

UML Class Diagrams for Inheritance Hierarchies

Figure 13.9 shows a UML class diagram for an inheritance hierarchy that keeps track of people associated with a department store. The top class, Person, is generic. It contains data and methods that are common to all classes in the hierarchy. Classes below the top class are more specific. For example, the Customer and Employee classes describe specific types of people in the department store. Because there are two distinct types of store employees, the Employee class has two subordinate classes for the two types—the FullTime class for full-time employees and the PartTime class for part-time employees.

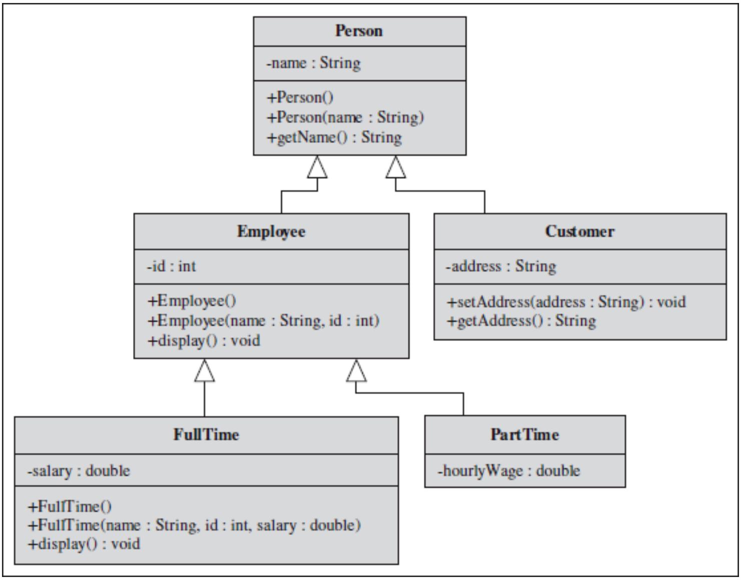

Person
-name : String
+Person()
+Person(name : String)
+getName() : String
Employee
-id : int
+Employee()
+Employee(name : String, id : int)
+display() : void
Customer
-address : String
+setAddress(address : String) : void
+getAddress() : String
FullTime
-salary : double
+FullTime()
+FullTime(name : String, id : int, salary : double)
+display() : void
PartTime
-hourlyWage : double

Figure 13.9 UML class diagram for a Person class inheritance hierarchy

Within an inheritance hierarchy, lower classes inherit upper classes' members. Thus, the Employee and Customer classes inherit name from the Person class. Likewise, the FullTime and PartTime classes inherit id from the Employee class. Inheritance travels all the way down the inheritance hierarchy tree, so in addition to inheriting id from the Employee class, the FullTime and PartTime classes inherit name from the Person class.

Within an inheritance hierarchy, classes are linked in pairs. Can you identify the linked pairs in Figure 13.9? The four pairs of linked classes are Person-Customer, Person-Employee, Employee-FullTime, and Employee-PartTime. For each pair of linked classes, the more general class is considered to be the *superclass,* and the more specific one is considered to be the *subclass*.

Inheriting a superclass's variables and methods enables a subclass to be a clone of its superclass. But making a subclass that's just a clone would be silly because you could just use the superclass instead. You always want a subclass to be a more specific version of its superclass. That's achieved by establishing additional variables and/or methods inside the subclass's definition. For example, in Figure 13.9, the Customer class defines an address instance variable. That means Customer objects have a name (inherited from the Person class) plus an address. Customer addresses are important in that they enable department stores to mail monthly “Everything-Must-Go Liquidation Sale!” advertisements to their customers.

UML class diagrams usually show superclasses above subclasses. However, that’s not always the case. With large projects, you’ll have lots of classes and several different types of relationships among the classes. With all that going on, it’s sometimes impossible to draw a “clean” class hierarchy picture and preserve the traditional superclass-above-subclass layout. Thus, subclasses sometimes appear at the left, at the right, or even above their superclasses. So how can you tell which is the subclass and which is the superclass? UML class diagrams use a solid line and a hollow arrow for inheritance relationships, with the arrow pointing to the superclass. In Figure 13.9, note how the arrows do indeed point to the superclasses.

Inheritance Terminology

Unfortunately, the terms superclass and subclass can be misleading. The "super" in superclass seems to imply that superclasses have more capability, and the "sub" in subclass seems to imply that subclasses have less capability. Actually, it's the other way around—subclasses have more capability. Subclasses can do everything that superclasses can do, and more.

For the most part, we'll stick with the terms superclass and subclass since those are the formal terms used by the designers of the Java language, but be aware that there is alternative terminology. Programmers often use the terms *parent class* or *base class* when referring to a superclass. And they often use the terms *child class* or *derived class* when referring to a subclass. The parent-child relationship between classes is important because it determines inheritance. With a human parent-child relationship, the child normally inherits physical features from the parent.1 The class parent-child relationship parallels the human parent-child relationship. But with a class parent-child relationship, the child doesn't inherit money; instead, it inherits the variables and methods defined in the superclass.

There are two more inheritance-related terms that you should be aware of. An *ancestor* class refers to any of the classes above a particular class in an inheritance hierarchy. For example, in Figure 13.9's inheritance hierarchy, Employee and Person are the ancestors of FullTime. A *descendant* class refers to any of the classes below a particular class in an inheritance hierarchy. For example, in Figure 13.9's inheritance hierarchy, Employee, Customer, FullTime, and PartTime are descendants of Person.

Benefits of Inheritance

Long before reading this chapter, you were already convinced of the benefit of modeling your programs with classes, right? (In case you need to be reminded why you love classes so much, it's because classes allow you to encapsulate things.) So you should be able to see the benefit of having a Customer class and also an Employee class for a department store program. OK, having separate Customer and Employee classes is good, but why stir up trouble and give them a superclass? If there's no superclass for the

Customer and Employee classes, then the things common to customers and employees would have to be defined in both classes. For example, you'd need a name instance variable and a getName method in both classes. But redundant code is almost always a bad idea. Why? With redundant code, debugging and upgrading chores are more tedious. After fixing or improving the code in one place, the programmer must remember to fix or improve the code in the other place as well.

In Figure 13.9, notice that classes at different levels in the hierarchy contain different instance variables, and they have different methods (although Employee and FullTime both have a display method, the methods are different; that is, they behave differently). There is no functional duplication, and there is maximal *code reusability*. Code reusability is when you have code that provides functionality for more than one part of a program. Putting common code from two classes into a superclass is one example of code reusability. Code reusability can also take place when you want to add a significant chunk of functionality to an existing class. You might want to implement the functionality by adding code directly to the existing class. But suppose the class works perfectly, and you're scared to touch it for fear of messing it up. Or maybe your know-it-all coworker wrote the class, and you don't want to risk getting him or her riled up over code modifications. No problem. Extend the class (i.e., create a subclass) and implement the new functionality in the extended class.

Don't try to change an ancestor.

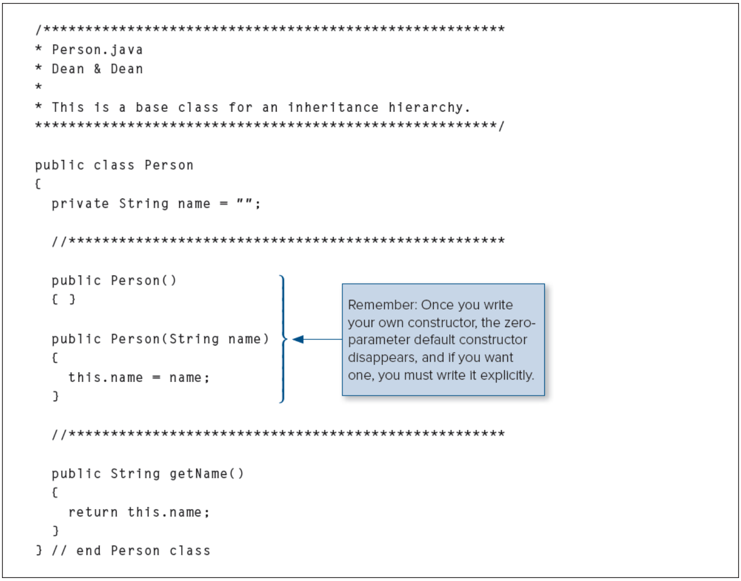
You've seen that inheritance gives rise to code reusability, and now you should be properly convinced of the benefits of code reusability. Another benefit of inheritance is that it gives rise to smaller modules (because classes are split into superclasses and subclasses). In general, smaller modules are good because there's less code to wade through when searching for bugs or making upgrades.

13.4 Implementation of a Person/Employee/FullTime Hierarchy

To explain how to implement inheritance, we'll implement the Person/Employee/FullTime hierarchy shown in Figure 13.9. We'll implement the Person and Employee classes in this section, and we'll implement the FullTime class in Section 13.6.

The Person class

Figure 13.10 contains an implementation of the Person class. It will be a superclass, but there's no special code in the Person class indicating that it will be a superclass. The special code comes later, when we define Person's subclasses. That's where we indicate that Person is a superclass for those subclasses.



```
/*******************************************************
* Person.java
* Dean & Dean
*
* This is a base class for an inheritance hierarchy.
*******************************************************/

public class Person
{
  private String name = "";

  //*****************************************************

  public Person()
  { }

  public Person(String name)
  {
    this.name = name;
  }

  //*****************************************************

  public String getName()
  {
    return this.name;
  }
} // end Person class
```

Remember: Once you write your own constructor, the zero-parameter default constructor disappears, and if you want one, you must write it explicitly.

Figure 13.10 Person class, a superclass for the Employee class

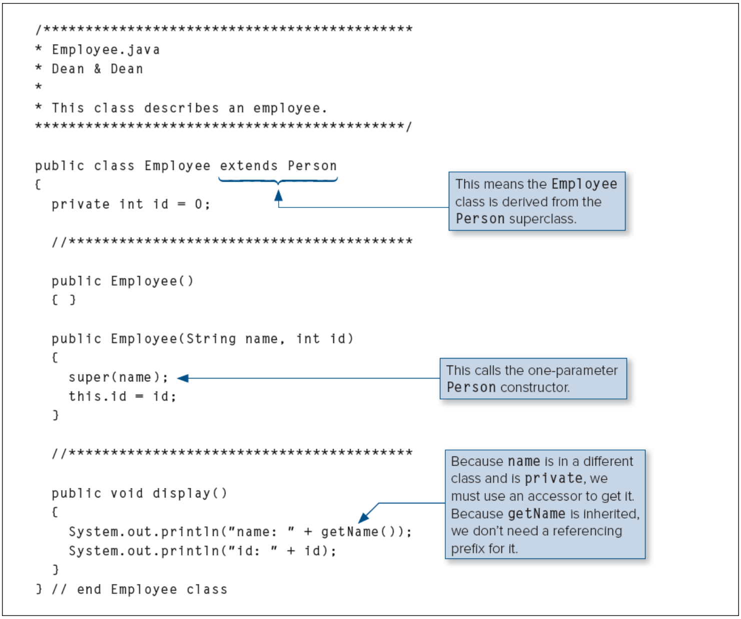
The Person class doesn't do much. It just provides a constructor that stores a name and an accessor that retrieves it.

However, it contains one item worth examining—the zero-parameter constructor. Normally, when a driver instantiates a Person class, the driver will assign the person's name by passing a name argument to the one-parameter constructor. But suppose you want to test your program with a Person object, and you don't want to hassle with storing a name in the Person object. The zero-parameter constructor allows you to do that. Do you know what name will be given to a Person object created by the zero-parameter constructor? In this case, name is a string instance variable, and the default value for a string instance variable is null. To avoid the ugly null default, note how name is initialized to the empty string.

Quick quiz: Can you achieve the same functionality by omitting the zero-parameter constructor because the compiler automatically provides a default zero-parameter constructor? Nope—remember that once you write any constructor, the compiler no longer provides a default zero-parameter constructor.

The Employee Class

Figure 13.11 contains an implementation of the derived Employee class, which provides an id. Note the extends clause in the Employee class's heading. To enable inheritance, an extends clause must appear at the right of the subclass's heading. Thus, extends Person appears at the right of the Employee class's heading. Note that the Employee class defines just one instance variable, id. Does that mean that an Employee object has no name? No. Employee objects do have names because the - Employee class inherits the name instance variable from the Person superclass. Now you'll learn how to access name from within the Employee class.



```
/*********************************************
* Employee.java
* Dean & Dean
*
* This class describes an employee.
*********************************************/

public class Employee extends Person
{
  private int id = 0;

  //*****************************************

  public Employee()
  { }

  public Employee(String name, int id)
  {
    super(name);
    this.id = id;
  }

  //*****************************************

  public void display()
  {
    System.out.println("name: " + getName());
    System.out.println("id: " + id);
  }
} // end Employee class
```

This means the `Employee` class is derived from the `Person` superclass.

This calls the one-parameter `Person` constructor.

Because `name` is in a different class and is `private`, we must use an accessor to get it. Because `getName` is inherited, we don't need a referencing prefix for it.

Figure 13.11 Employee class, derived from the Person class

The Employee class's display method is in charge of printing an employee's information—name and id. Printing the id is easy because id is declared within the Employee class. Printing name requires a bit more work. Since name is a private instance variable in the Person superclass, the Employee class cannot access name directly (that’s the same interpretation of private that we’ve always had). But the Employee class can access name by calling the Person class’s public getName accessor method. Here’s the relevant code from the display method:

```
System.out.println("name: " + getName());
```

As you might recall, in an instance method, if you call a method that’s in the same class as the class you’re currently in, the reference variable dot prefix is unnecessary. Likewise, in an instance method, if you call a method that’s in the superclass of the class you’re currently in, the reference variable dot prefix is unnecessary. Thus, there’s no reference variable dot prefix in this call to getName.

While there is complete agreement about the way that a subclass object can access a private instance variable from its superclass (i.e., use a superclass public accessor method), there is disagreement about the terminology used to describe such access. Some textbook authors say that a subclass object inherits private instance variables from its superclass; others say that a subclass object does not inherit private instance variables from its superclass. We fall into the inheritance camp because an instantiated subclass contains private instance variables from the superclass. We know this because they are accessible (via public methods) without having to instantiate a second object for the superclass. In an explanation that would make a politician proud, Oracle first claims that private members are inherited, and then later claims that private members are not inherited (see what we mean by looking at the statement, at https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html). No worries. The terminology doesn't matter, as long as you know how things work in terms of functionality.

13.5 Constructors in a Subclass

Let's now examine Figure 13.11's two-parameter Employee constructor. The goal is to assign the passed-in name and id values to the associated instance variables in the instantiated Employee object. Assigning to the id instance variable is easy because id is declared within the Employee class. But assigning to the name instance variable is harder because name is a private instance variable in the Person superclass. There's no setName mutator method in Person, so how does name get set? Read on. . . .

Using super to Call a Superclass Constructor

Employee objects inherit the name instance variable from Person. It follows that Employee objects should use the Person constructor to initialize their inherited name instance variables. But how can an Employee object call a Person constructor? It's easy—once you know how. To call a superclass constructor, use the reserved word super followed by parentheses and a comma-separated list of arguments that you want to pass to the constructor. For example, here's how Figure 13.11's Employee constructor calls the one-parameter Person constructor:

```
super(name);
```

Calls to super are allowed only in one particular place. They're allowed only from within a constructor, and they must be the first line within a constructor. That should sound familiar. In Chapter 7, you learned another usage for the keyword this, a usage that is distinct from using this dot to specify an instance member. The syntax for this other usage of this is:

this(*arguments*);

This kind of this usage calls another (overloaded) constructor from within a constructor in the same class. And recall that you must make such a call on the first line of your constructor.

By the way, would it be legal to have a this constructor call and a super constructor call within the same constructor? No, because with both constructor calls in the same constructor, that means only one of the constructor calls can be in the first line. The other one would violate the rule that constructor calls must be in the first line.

Default Call to Superclass Constructor

Java developers like to call superclass constructors because doing so promotes software reuse. If you write a subclass constructor and don't include a call to another constructor (with this or with super), the Java compiler sneaks in and inserts a superclass zero-parameter constructor call by default. Thus, although Figure 13.11's Employee zero-parameter constructor has an empty body, the Java compiler automatically inserts super(); in it. So these two constructors are functionally equivalent:

```
public Employee()
{ }

public Employee()
{
  super();
}
```

The explicit super(); call makes it clear what's going on. Feel free to include it if you wish, to make your code more self-documenting.

Whenever a constructor is called, the JVM automatically runs up the hierarchical tree to the greatest-grandparent's constructor, and it executes that greatest-grandparent's constructor first. Then it executes the code in the constructor below it, and so on, and finally, it executes the rest of the code in the originally called constructor.2

13.6 Method Overriding

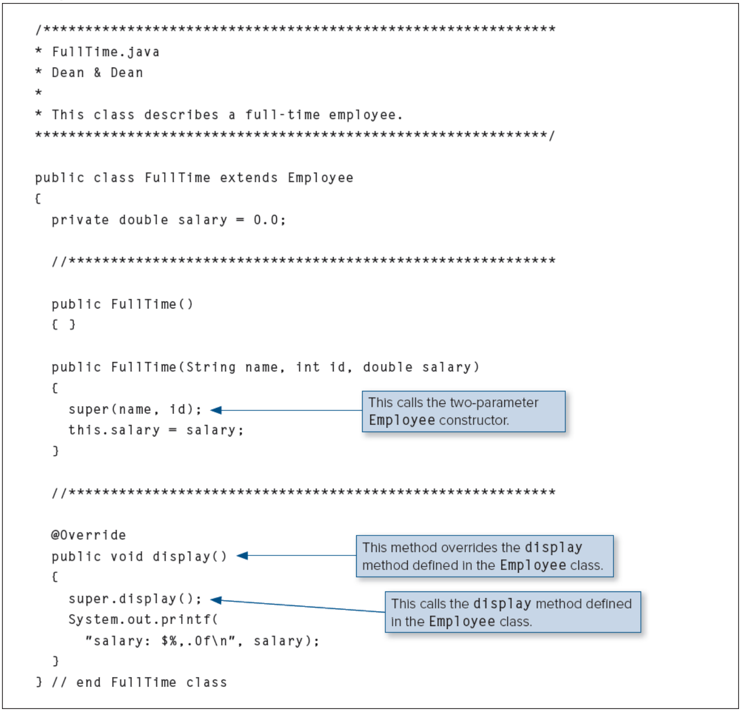
From Chapter 7, you know about method overloading. That's when a single class contains two or more methods with the same name but a different sequence of parameter types—approximately the same semantics, but different syntax. Now for a related concept—*method overriding*. That's when a subclass has a method with the same name, the same sequence of parameter types, and the same return type as a method in a superclass. An overriding method has the same syntax but different semantics. The term "overriding" should make sense when you realize that an overriding method overrides/supersedes its associated - superclass method. That means, by default, an object of the subclass uses the subclass's overriding method and not the superclass's overridden method.

The concept of a subclass object using the subclass's method rather than the superclass's method falls in line with this general principle of programming: Local stuff takes precedence over global stuff. Can you think of where else this rule applies? If a local variable and an instance variable have the same name, the local variable takes precedence when you're inside the local variable's method. The same reasoning applies to parameters taking precedence over instance variables when you're inside the parameter's method.

Method Overriding Example

To explain method overriding, we'll continue with the implementation of the Person/Employee/FullTime program. We implemented the Person and Employee classes in Section 13.4. We implement the FullTime class in Figure 13.12. Note FullTime's display method. It has the same sequence of parameter types as the display method in the Employee class of Figure 13.11. Because the FullTime class extends the Employee class, the FullTime class's display method overrides the Employee class's display method. The Java annotation, @Override, asks the compiler to confirm that the method is indeed an overriding method. This is not required, but it provides documentation and helps avoid misunderstanding.

Page 608



```
/*******************************************************************
* FullTime.java
* Dean & Dean
*
* This class describes a full-time employee.
*******************************************************************/

public class FullTime extends Employee
{
  private double salary = 0.0;

  //*****************************************************************

  public FullTime()
  { }

  public FullTime(String name, int id, double salary)
  {
    super(name, id);
    this.salary = salary;
  }

  //*****************************************************************

  @Override
  public void display()
  {
    super.display();
    System.out.printf(
      "salary: $%,.0f\n", salary);
  }
} // end FullTime class
```

This calls the two-parameter `Employee` constructor.

This method overrides the `display` method defined in the `Employee` class.

This calls the `display` method defined in the `Employee` class.

Figure 13.12 FullTime class, which illustrates method overriding

Using super to Call an Overridden Method

Don't
re-invent
the wheel.

When you override a method, you still

can use the overridden method for the services it provides and then provide new code only for additional services. To do this, start your new method with a statement that calls the overridden method by prefixing the overridden method name with super dot. During program development, you can tentatively apply @Override annotations just above new method headers to see if ancestral methods exist that might already provide some of the services you need. Then, if the compiler does not find an overridden method, it will tell you that, and you can discard your tentative @Override annotation.

For example, in Figure 13.12's FullTime subclass, note how the display method calls the superclass's display method with super.display();.

Now look again at that super.display() method call in Figure 13.12's FullTime class. What do you suppose would happen if you forgot to prefix that method call with super dot? Without the prefix, ⚠ display(); would call the display method in the current class, FullTime, not the display method in the superclass. In executing the FullTime class's display method, the JVM would call the FullTime class's display method again. This process would repeat in an infinite loop.

By the way, you can have a series of overriding methods; that is, you can override an overriding method. But it's illegal to have a series of super dot prefixes chained together. In other words, in the Person/Employee/FullTime inheritance hierarchy, suppose the Person class contains a display method that's overridden by the Employee and FullTime classes. In the FullTime class, it would be illegal to call the Person class's display method like this:

```
super.super.display();
```

compilation error

To call the Person class's display method from the FullTime class, you'd have to call the Employee class's display method and rely on the Employee class's display method to call the Person class's display method.

Have you noticed that super has two different purposes? You can use super dot to call an overridden method, and you can also use super with parentheses as in super(name) to call a superclass's constructor.

Return Types Must Be the Same

⚠ An overriding method must have the same return type as the method that it's overriding. If it has a different return type, the compiler generates an error. Said another way, if a subclass and a superclass have methods with the same name, the same sequence of parameter types, and different return types, the compiler generates an error.

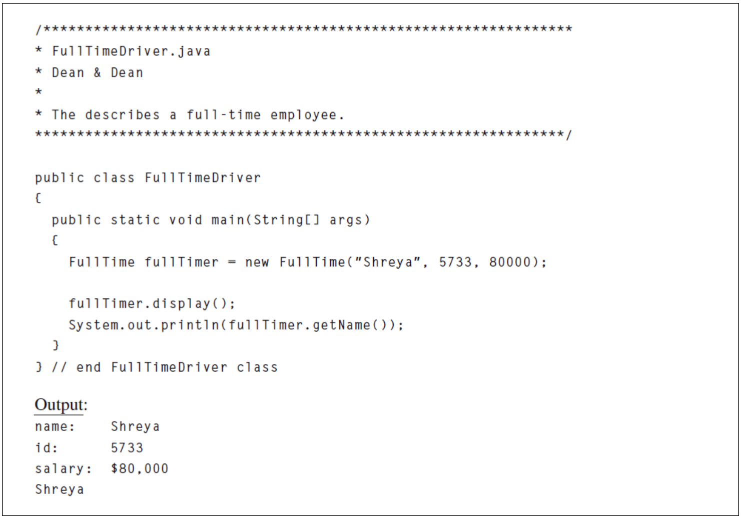
This error doesn't occur all that

often because if you’ve got methods with the same names and sequences of parameter types, you’ll usually also want the same return types. But you’ll see the error crop up every now and then when you’re debugging, so be aware of it. By the way, if a subclass and a superclass have methods with the same name and different sequences of parameter types, it doesn’t matter if the return types are the same. Why? Because such methods are not in an overriding relationship. They are different methods entirely.

13.7 Using the Person/Employee/FullTime Hierarchy

Now let’s reinforce what you’ve learned about inheritance by looking at what happens when you instantiate an object of the lowest-level derived type and use that object to call overridden methods and inherited methods. Figure 13.13 contains a driver for the FullTime class, and the subsequent output shows what it does. This driver instantiates a fullTimer object from the FullTime class. Then the fullTimer object calls its display method. As shown in Figure 13.12, this display method uses super to call the Employee class's display method, which prints the fullTimer's name and id. Then fullTimer's display method prints the fullTimer's salary.

In the final statement in Figure 13.13, the fullTimer object calls its getName method and prints fullTimer's name. But wait a minute! The FullTime class does not have a getName method, and its superclass, Employee, does not have one either. The code seems to be calling a nonexistent method. What's going on here? What's going on is inheritance—produced by those wonderful little extends clauses. Because there is no explicitly defined getName method in its own FullTime class, the fullTimer object goes up its inheritance hierarchy until it finds a getName method, and then it uses that method. In this case, the first getName method found is in the Person class, so that's the method the fullTimer object inherits and uses. There is no need to use super dot to access the getName method (but using super dot would work, in case you're curious). If a method is not in the current class, the JVM automatically goes up the inheritance hierarchy and uses the first definition of that method it finds.



```
/*************************************************************
* FullTimeDriver.java
* Dean & Dean
*
* The describes a full-time employee.
*************************************************************/

public class FullTimeDriver
{
  public static void main(String[] args)
  {
    FullTime fullTimer = new FullTime("Shreya", 5733, 80000);

    fullTimer.display();
    System.out.println(fullTimer.getName());
  }
} // end FullTimeDriver class
```

Output:

```
name:    Shreya
id:      5733
salary:  $80,000
Shreya
```

Figure 13.13 Driver of constructors and methods in an inheritance hierarchy

Page 610

Notice that our driver did not instantiate any Employee or Person objects. It just instantiated an object from a class at the bottom of the inheritance hierarchy only. This is the way a good inheritance hierarchy should be used. Ideally, you should just instantiate objects from classes at the bottom of the hierarchy. Ideally, all the classes above the bottom classes are there to make the bottom classes simple. In real life, we often do use classes above the bottom, but using bottom classes only is the ideal situation.

13.8 The final Access Modifier

You've used the final access modifier for quite a while now to turn a variable into a named constant. In this section, you'll learn how to use final to modify a method and modify a class.

If you use the final modifier in a method heading, you'll prevent the method from being overridden with a new definition in a subclass. You might want to do this if you think that your method is perfect and you don't want its original meaning to "drift." You might also want to consider using final to help speed things up a bit. Methods that use the final modifier should run faster because the compiler can generate more efficient code for them. The code efficiency comes from the compiler not having to prepare for the possibility of inheritance. However, the speed improvement is miniscule for adding final to a single method, and you probably won't notice it unless you have a large programming project with lots of subclasses and you use final a lot.

If you use the final access modifer in a class heading, you prevent the class from having any subclasses. You might want to do this if you have a class that's good and reliable, and you want to preserve its quality and protect it from future "feature creep." By the way, if a class is declared to be a final class, there's no point in specifying final for any of its methods. A final class cannot be extended, so overriding methods cannot exist. The final modifier helps prevent invasion by hackers.

Even though it may be difficult to see palpable benefits from the use of final, you should use it to improve security. Even if you don't use it for your own programs, you'll need to understand it because you'll see it quite often in the Java API library classes. For example, the Math class is defined with the final access modifier, so it's illegal to extend the Math class and override any of its methods.

13.9 Using Inheritance with Aggregation and Composition

We have described several ways classes can be related—with aggregation, composition, and inheritance. Now let's consider using all three relationships together.

Aggregation, Composition, and

Inheritance Compared

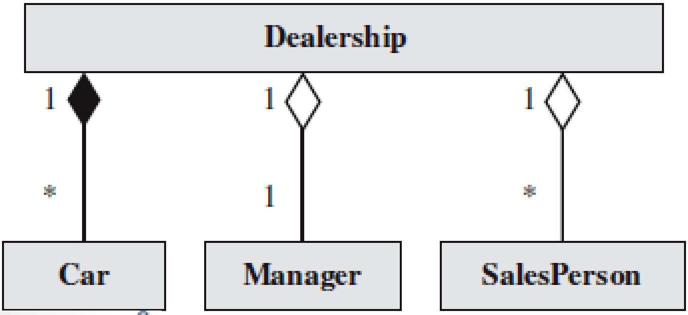
Aggregation and composition both implement a has-a relationship. We call aggregation and composition relationships *has-a* relationships because one class, the container class, has a component class inside it. For example, in Section 13.2's Dealership program, a dealership has a sales manager, with nonexclusive ownership rights, and that's why the Dealership program implements the Dealership-SalesManager relationship with aggregation. Also, a dealership has an inventory of cars, with exclusive ownership rights, and that's why the Dealership program implements the Dealership-Car relationship with composition.

Inheritance implements an *is-a* relationship. We call an inheritance relationship an is-a relationship because one class, a subclass, is a more detailed version of another class. For example, in the Person/ Employee/ FullTime program, a full-time employee is an employee, and that's why the program implements the FullTime-Employee relationship with inheritance. Also, an employee is a person, and that's why the program implements the Employee-Person relationship with inheritance as well.

It's important to keep in mind that these are not alternative ways to represent the same relationship. They are ways to represent different relationships. The aggregation and composition relationships are when one class is a whole made up of nontrivial constituent parts defined in other classes. The inheritance relationship is when one class is a more detailed version of another class. More formally, inheritance is when one class, a subclass, inherits variables and methods from another class, a superclass, and then supplements those with additional variables and methods. Because composition and inheritance deal with different aspects of a problem, many programming solutions include a mixture of both paradigms.

Aggregation, Composition, and Inheritance Combined

In the real world, it’s fairly common to have aggregation, composition, and inheritance relationships together in the same program. Let’s look at an example that uses all three class relationships. Section 13.2’s Dealership program uses aggregation and composition, and we repeat it here, with the non-Dealership aggregation relationships omitted for simplicity.



Factor out the common code.

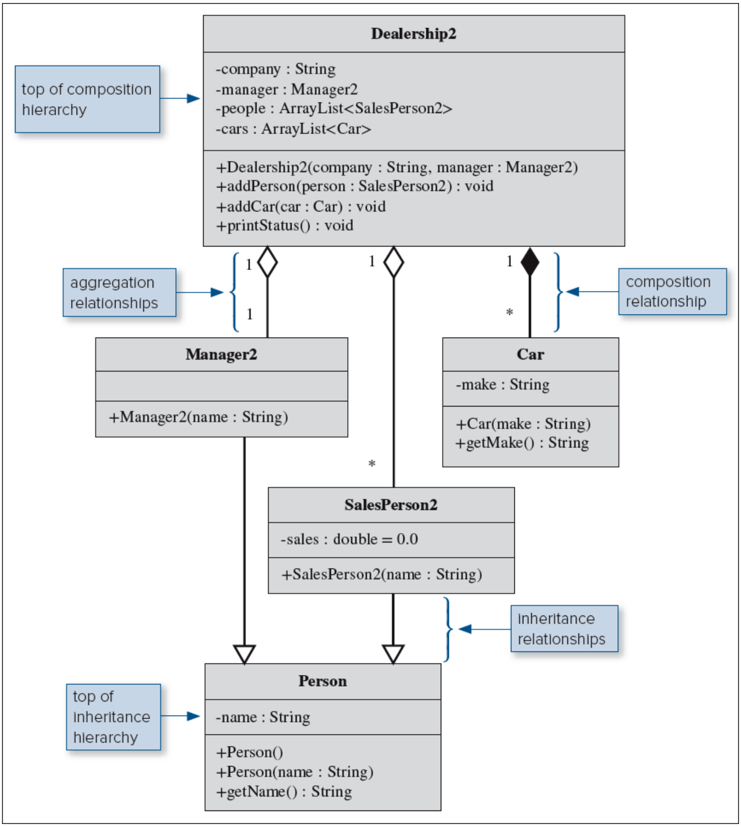
What sort of inheritance relationship could or should be added to the Dealership program? If you look back at Figures 13.5 (the Manager class) and 13.6 (the SalesPerson class), you'll see that Manager and SalesPerson both declare the same instance variable, name, and they both define the same instance method, getName. That's an example of undesirable duplication, and we can use inheritance to eliminate that duplication. Introducing inheritance into that program does not alter the original whole-parts structure. It just introduces a complementary mechanism that eliminates duplication.

Figure 13.14 shows an improved and expanded UML class diagram for a new

Dealership2 program. If you compare this with the previous UML class diagram, you'll see that each class is fleshed out to include instance variables and methods. Figure 13.14's diagram also includes a Person class. Our previous Manager and SalesPerson classes now inherit a variable, two constructors, and a method from this Person class. The inheritance reduces the Manager and SalesPerson classes to the simpler Manager2 and SalesPerson2 classes. These simpler classes do not need explicit declaration of name and explicit definition of getName because they inherit these members from Person. Read through the code for the shortened Manager2 and SalesPerson2 classes in Figures 13.15 and 13.16.

The Car class is unchanged from the original Dealership program; if you want to see its code, look back at Figure 13.4. The Dealership2 and

Dealership2Driver classes are the same as the Dealership and DealershipDriver classes defined in Figures 13.3 and 13.7, respectively, except that Dealership is changed to Dealership2, Manager is changed to Manager2, and SalesPerson is changed to SalesPerson2.

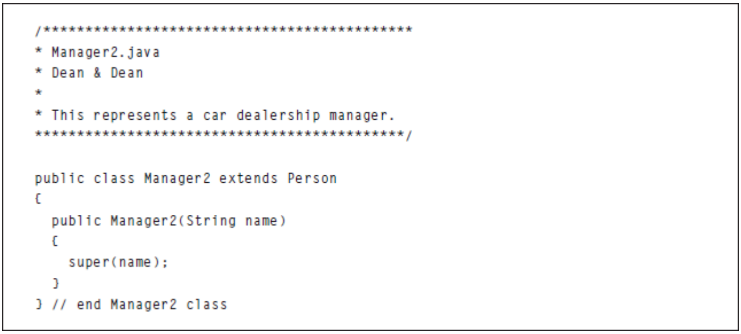


Dealership2
-company : String
-manager : Manager2
-people : ArrayList<SalesPerson2>
-cars : ArrayList<Car>
+Dealership2(company : String, manager : Manager2)
+addPerson(person : SalesPerson2) : void
+addCar(car : Car) : void
+printStatus() : void
top of composition hierarchy
aggregation relationships
composition relationship
1
1
1
1
*
*
Manager2
+Manager2(name : String)
Car
-make : String
+Car(make : String)
+getMake() : String
SalesPerson2
-sales : double = 0.0
+SalesPerson2(name : String)
inheritance relationships
top of inheritance hierarchy
Person
-name : String
+Person()
+Person(name : String)
+getName() : String

Figure 13.14 Class diagram for the revised car dealership program—

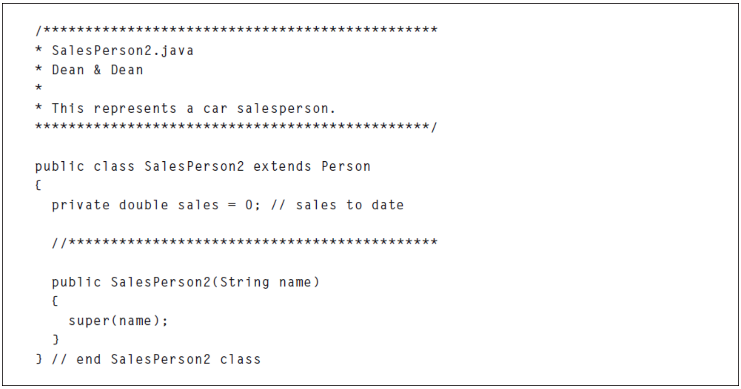
Dealership2

In Figure 13.14, the addition of the Person class makes it look as though we made the Dealership2 program bigger by adding another class. But the additional Person class was already defined in another program, the Person/Employee/FullTime program. In borrowing the Person class from that program, we got something for nothing. The borrowed Person class enabled us to shorten two other classes. Being able to borrow classes that have already been written and then inheriting from them in other contexts is an important benefit of OOP. If you look at the prewritten classes in the Java API, you'll see that they do a lot of inheriting from one to another, and in many cases, you have the option of inheriting from them into your own programs as well.



```
/********************************************
* Manager2.java
* Dean & Dean
*
* This represents a car dealership manager.
********************************************/

public class Manager2 extends Person
{
  public Manager2(String name)
  {
    super(name);
  }
} // end Manager2 class
```

Figure 13.15 Manager2 class for the Dealership2 program



```
/***********************************************
* SalesPerson2.java
* Dean & Dean
*
* This represents a car salesperson.
***********************************************/

public class SalesPerson2 extends Person
{
  private double sales = 0; // sales to date

  //********************************************

  public SalesPerson2(String name)
  {
    super(name);
  }
} // end SalesPerson2 class
```

Figure 13.16 SalesPerson2 class for the Dealership2 program

13.10 Design Practice with Card Game Example

Learn by doing.

Page 613

In the previous section, you learned how to use different types of class relationships together in a single program. The way you learned was adding inheritance to an existing program. In this section, you'll once again use different types of class relationships, but this time you'll design the program from the ground up. And you'll be doing most of the work, rather than just understanding how it's done by someone else.

Your Mission (Should You Choose to Accept It)

Your mission is to design and implement a generic card game program. In carrying out this mission, follow these guidelines:

- Assume it's a game like War or Gin Rummy, where you have a deck of cards and two players.
- Decide on appropriate classes.

For each class, draw a UML class diagram and write in the class name.

- Look for composition relationships between classes. For each pair of classes related by composition, draw a compositional association line with appropriate multiplicity values.
- For each class, decide on appropriate instance variables.
- For each class, decide on appropriate public methods.
- Look for common instance variables and methods. If two or more classes contain a set of common instance variables and methods, provide a superclass and move the common instance variables and methods to the superclass. The classes originally containing common members now become subclasses of the

superclass. For each subclass-superclass pair, draw an association line with an inheritance arrow from the subclass to the superclass to indicate an inheritance relationship.

Now go ahead and use these guidelines to draw a UML class diagram for a generic card game program. Because this is a nontrivial exercise, you may be tempted to look at our solution before trying to come up with a solution on your own. Please resist that temptation! By implementing your own solution, you'll learn more and make yourself aware of potential problems.

Defining the Classes and the Relationships Between Them

Have you finished your class diagram?

If so, then you may continue. . . .

In coming up with a class diagram, the first thing to do is to decide on the classes themselves. Unfortunately, that’s a bit of an art. The easy classes are the ones that directly correspond to something you can see. In visualizing a card game, can you see two people holding cards and taking additional cards from a deck that sits between them? You should be able to see a deck, two hands, individual cards, and two people. For the deck, use a Deck class. For the two hands, use a Hand class. For the individual cards, use a Card class. You may or may not wish to represent the people. If you’re implementing an elaborate card game where players have personalities, use a Person class. Otherwise, there’s no need for a Person class. Let’s keep things simple and not implement a Person class.

In thinking about the big picture,

you should ask yourself, “What is a game?” A game is a composition of several parts, so define Game as a whole class and define other classes as the parts of the game. A Game is composed of three components/parts—a deck and two hands. Thus, Deck and Hand are part classes within the Game composition class. In Figure 13.17’s class diagram, note the association line connecting Game to Deck. The association line has a solid diamond, which indicates composition, and it has 1-to-1 multiplicity values, which indicate each game has one deck. The Game to Hand association line also has a solid diamond for composition, but it has 1-to-2 multiplicity values, which indicate each game has two hands.

Coming up with the idea of using a Game class is probably more difficult than coming up with the ideas for using Deck, Hand, and Card classes. Why? A game is nontactile (that is, you can't touch it), so it's hard to see it as a class. Why bother with having a Game class? If you omit the Game class, you could still implement a card game. Instead of declaring Deck and Hand objects inside the Game class, you could declare them inside the main method. But it's more elegant to put them inside a Game class. Why? By putting them inside a Game class, it furthers the goal of encapsulation. Also, it enables main to be streamlined. As you'll see later on, if you have defined a Game class, the driver's main method just needs to instantiate a Game object and then call playAGame—that's it. You can't get much more streamlined (and elegant) than that.

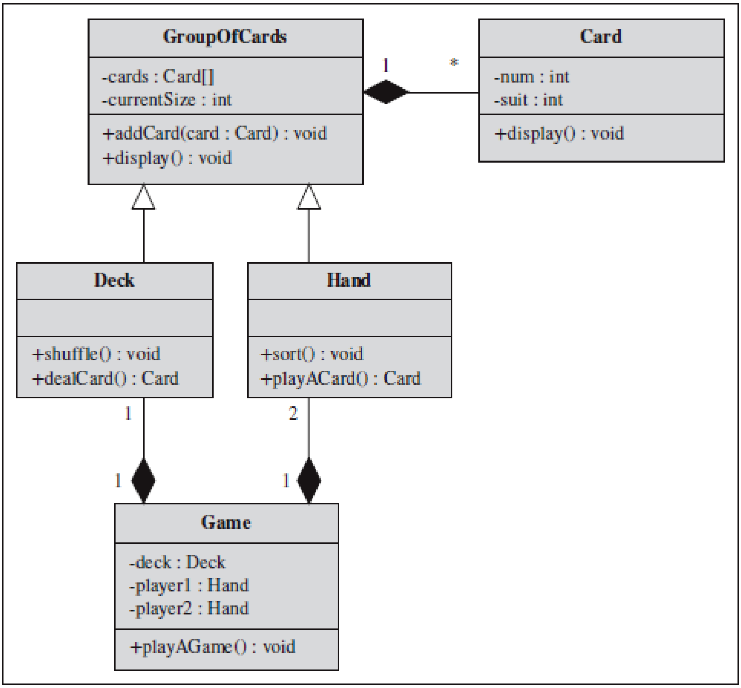

GroupOfCards
-cards : Card[]
-currentSize : int
+addCard(card : Card) : void
+display() : void
1
*
Card
-num : int
-suit : int
+display() : void
Deck
+shuffle() : void
+dealCard() : Card
Hand
+sort() : void
+playACard() : Card
1
2
1
1
Game
-deck : Deck
-player1 : Hand
-player2 : Hand
+playAGame() : void

Figure 13.17 Preliminary class diagram for a Card Game program

For each class in the Card Game program, what are its members (that is, instance variables and methods)? Let's tackle the easy classes first—Game and Card. The Game class needs three instance variables—one for the deck and two for the two hands. It needs a method for playing a game. The Card class needs two instance variables—one for a number (two through ace) and one for a suit (clubs through spades). It needs a method to display the card's number and suit values. As a sanity check, verify that Figure 13.17's Game and Card members match what we've described.

The Deck class needs an instance variable for an array of cards such that each card is a Card object. The Deck class also needs an instance variable to keep track of the current size of the deck. The Deck class needs methods for shuffling and dealing. To help with debugging, you should probably also include a method to display all the cards in the deck.

The Hand class needs instance variables for an array of cards and for a current-size value. It needs methods for displaying all the cards, adding a card to the hand, and playing a card from the hand. For most card games, you'd also want a method to sort the hand. Different card games would use different and/or additional Hand methods. We'll keep things simple and not worry about them.

The next step is to try to identify common members and move them to a superclass. The Deck and Hand classes have three common members—a cards array variable, a currentSize variable, and a display method. In moving those members to a superclass, what would be a good name for such a class? It should be something generic that can be used as the superclass for both Deck and Hand. Either GroupOfCards or just plain Cards sounds pretty good. Let's use GroupOfCards. In Figure 13.17's class diagram, note the inheritance association lines connecting Deck and Hand to GroupOfCards.

We've now examined the members in all five classes in the Card Game program, and we've examined the relationships between four of the classes—Game, Deck, Hand, and GroupOfCards. The last piece of the UML-class-diagram puzzle is the relationship between GroupOfCards and Card. Is it an is-a or a has-a relationship? It’s not an is-a relationship because it doesn’t make sense to say that a group of cards is a card or a card is a group of cards. Instead, it’s a has-a relationship because a group of cards has a card (a group of cards usually has more than one card, but that doesn’t negate the has-a relationship). In Figure 13.17, note the has-a composition association line connecting GroupOfCards to Card. Figure 13.17 suggests implementing the composition as an array called cards, but it could be an ArrrayList.

Design is an iterative process.

Note that Figure 13.17’s label says “preliminary” class diagram. It’s preliminary because for a decent-sized application, it's nearly impossible to get the class diagram 100% right on your first-cut attempt. When you're done coding and testing(!) your prototype program, you should go back and update your class diagram appropriately. The class diagram serves two purposes. Early in the design process, it helps organize ideas and it keeps everybody on the same page. In the post-implementation phase, it serves as documentation that helps interested parties quickly get a handle on the application's organization.

Inheritance Versus Composition

When deciding on the relationship between two classes, it's usually pretty clear whether to use inheritance or composition. For example, in the Dealership program, a Manager is a Person, so inheritance is used. In the Card Game program, a Game has a

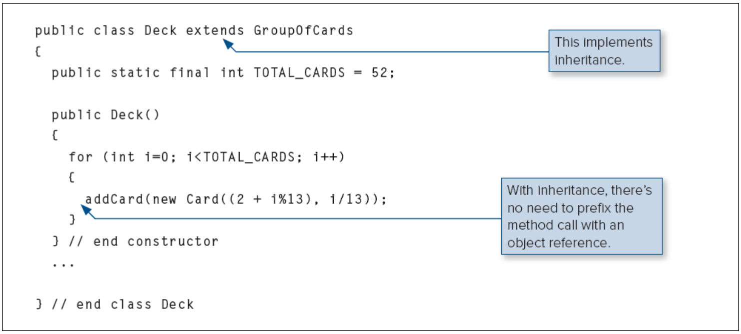
Deck, so composition is used.

However, sometimes it's not so clear-cut. For example, you could make the claim that a Deck is a GroupOfCards, and you could also make the claim that a Deck has a GroupOfCards. Where an inheritance is-a relationship and a composition has-a relationship both exist, opinions differ as to which strategy is better. Inheritance implementations tend to be more understandable. Composition implementations can be more encapsulated. (If you implement inheritance using the protected modifier for superclass data, the subclass will have direct access to the superclass data, and that breaks the superclass's encapsulation somewhat.) Let's compare the two strategies in the context of the Card Game program.

Figure 13.18's Deck class implements the relationship with inheritance. Figure 13.19's alternative

Deck class implements it with composition. We feel that Figure 13.18's inheritance code is more elegant than Figure 13.19's composition code. It has one less line, which is a good thing, but more importantly, it isn't cluttered with references to a groupOfCards variable. In the composition code, you're required to: (1) declare a groupOfCards variable, (2) instantiate the groupOfCards variable, and (3) prefix the call to addCard with the -groupOfCards calling object. Isn't the inheritance code nicer, where you don't have to worry about all that? In particular, you can call addCard directly (no groupOfCards calling object required), and that results in more readable code. By the way, the addCard method is defined in the GroupOfCards class. With inheritance, the fact that it's in a separate class from Deck is transparent. In other words, you call it from the Deck constructor the same way that you would call any other Deck method—without a calling object.

For some class pairs (like Deck and GroupOfCards), it's legal to use either an inheritance or a composition relationship. But it's never OK to use both inheritance and composition for the same feature. What would happen if Deck declared a GroupOfCards local variable and Deck also inherited from a GroupOfCards class? Deck objects would then contain two separate groups of cards and that's wrong!

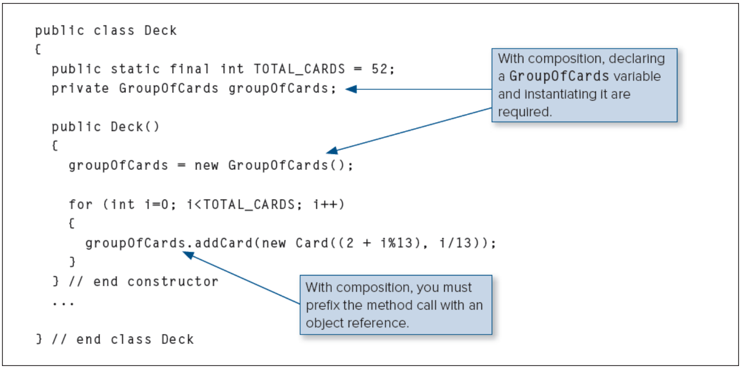


```
public class Deck extends GroupOfCards
{
  public static final int TOTAL_CARDS = 52;

  public Deck()
  {
    for (int i=0; i<TOTAL_CARDS; i++)
    {
      addCard(new Card((2 + i%13), i/13));
    }
  } // end constructor
  ...

} // end class Deck
```

This implements inheritance.

With inheritance, there's no need to prefix the method call with an object reference.

Figure 13.18 Inheritance implementation for the Deck class



```
public class Deck
{
  public static final int TOTAL_CARDS = 52;
  private GroupOfCards groupOfCards;

  public Deck()
  {
    groupOfCards = new GroupOfCards();

    for (int i=0; i<TOTAL_CARDS; i++)
    {
      groupOfCards.addCard(new Card((2 + i%13), i/13));
    }
  } // end constructor
  ...

} // end class Deck
```

With composition, declaring a `GroupOfCards` variable and instantiating it are required.

With composition, you must prefix the method call with an object reference.

Figure 13.19 Composition implementation for the Deck class

Designing is a gradual process.

At this point, you might want to go back to Figure 13.17's preliminary UML class diagram and add more detail. We didn't bother with constants or constructors in Figure 13.17's class diagram. In working with the Deck class skeleton (see Figure 13.18), it's now clear there's a need to (1) add a TOTAL_CARDS constant to the Deck class, (2) add a constructor to the Deck class, and (3) add a constructor to the Card class. For practice, we encourage you to update Figure 13.17's class diagram with these changes in mind. If you don't feel like it, that's OK; our main point here is to make you aware of the iterative nature of the program design process. Try to organize your thoughts as clearly as possible up front, but be prepared to adjust those thoughts later on.

Code to Get You Started

Once you've finished with the card game's class diagram, normally the next step would be to implement the classes with Java code. We won't bother to show class implementation details, but we would like to show you how the suggested classes might be driven by a main method. Having followed proper OOP design guidelines, it's easy to produce an elegant main method—see Figure 13.20. Note how short and understandable the main method is. Yeah!

```
public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  String again;
  Game game;

  do
  {
    game = new Game();
    game.playAGame();
    System.out.print(”Play another game (y/n)?: ”);
    again = stdIn.nextLine();
  } while (again.equals(”y”));
} // end main
```

Figure 13.20 Card Game program’s main method

Another example will illustrate further how the finished classes might be used by the rest of the program. See main’s call to playAGame in Figure 13.20. Figure 13.21 shows a partial implementation for the playAGame method. To shuffle the deck, call deck.shuffle(). To deal a card to the first player, call player1.addCard(deck.dealCard()). How’s that for straightforward?

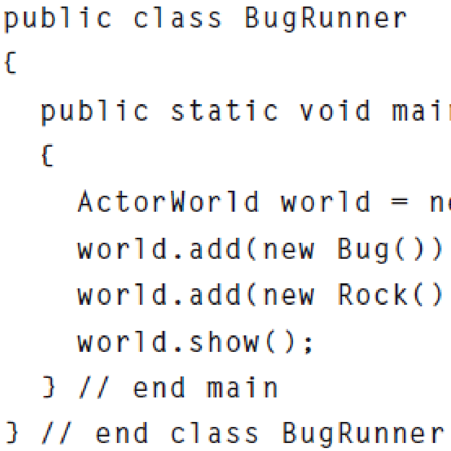
```
public void playAGame()
{
  deck.shuffle();

  // Deal all the cards to the two players.
  while (deck.getCurrentSize() > 0)
  {
    player1.addCard(deck.dealCard());
    player2.addCard(deck.dealCard());
  }
  ...

} // end playAGame
```

Figure 13.21 Partial implementation for the Game class's playAGame method

We'll leave it to you to finish this program. Two end-of-chapter exercises and a project suggest various elaborations.

13.11 GridWorld Case Study Extensions (Optional)

At the end of the previous chapter, in Section 12.17, we introduced the GridWorld case study. The GUI output in that section's Figure 12.18 was generated by running this driver3:



```
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.add(new Bug());
    world.add(new Rock());
    world.show();
  } // end main
} // end class BugRunner
```

In this code, each add method call selects a random location in the range of Figure 12.18's two-dimensional grid. Then it asks its parameter (Bug or Rock) to execute the putSelfInGrid method inherited from the Actor class. The putSelfInGrid method calls the put method in the BoundedGrid class. This put method inserts the Bug or Rock object into a two-dimensional occupantArray at the selected location.

The show method call instantiates a WorldFrame, which creates the GUI window. WorldFrame's constructor instantiates a GridPanel object. An argument passed to the WorldFrame constructor gives the WorldFrame object a reference back to the ActorWorld object that created it. The WorldFrame constructor uses this reference to get a reference to the BoundedGrid. The WorldFrame passes this BoundedGrid reference on to the new GridPanel. GridPanel's drawOccupants method uses this BoundedGrid reference to call BoundedGrid's getOccupiedLocations method. This method tells GridPanel's paintComponent method where to put Actor images each time it repaints the screen.

Actor Images

The Actor images come from .gif files, where "gif" stands for Graphic Interchange Format.4 Figure 13.22a shows the images contained in the framework/info/gridworld/actor subdirectory.

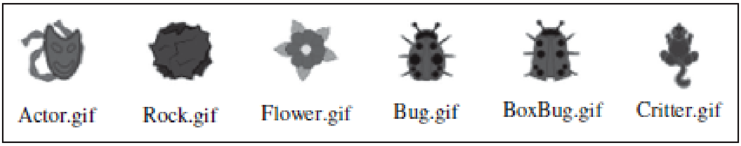

Actor.gif
Rock.gif
Flower.gif
Bug.gif
BoxBug.gif
Critter.gif

Figure 13.22a GridWorld images in framework/info/gridworld/actor

College Board

Figure 13.22b shows the images contained in the projects/critters subdirectory.

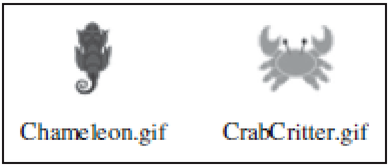

Chameleon.gif
CrabCritter.gif

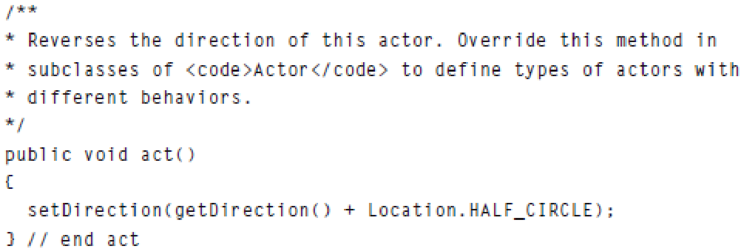
Figure 13.22b GridWorld images in projects/critters

College Board

GridWorld's GUI code initially assumes that the appropriate image filename is the corresponding actor's class name followed by .gif. If it can't find a .gif file starting with that name in the same directory as the actor's class, it goes up to that actor's superclass and looks in that superclass's directory for that superclass name followed by .gif, and so forth, up the actor's inheritance hierarchy.

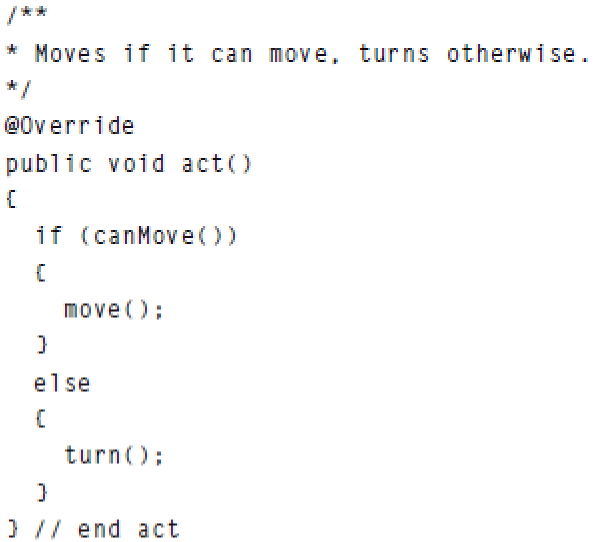
Actor Behavior

When it comes to actor behavior, GridWorld's pivotal method is the act method. In the base class for actors, Actor, the act method doesn't do much. It just reverses the actor's direction, like this:



```
/**
* Reverses the direction of this actor. Override this method in
* subclasses of <code>Actor</code> to define types of actors with
* different behaviors.
*/
public void act()
{
  setDirection(getDirection() + Location.HALF_CIRCLE);
} // end act
```

But this is not how a Bug in the BugRunner program acts. In that program, each bug moves forward until it hits a grid boundary. Then, if not impeded by rocks, it runs clockwise around the grid's boundary. To get this behavior, the Bug class extends the Actor class and overrides the act method like this:



```
/**
* Moves if it can move, turns otherwise.
*/
@Override
public void act()
{
  if (canMove())
  {
    move();
  }
  else
  {
    turn();
  }
} // end act
```

In GridWorld's projects/boxBug directory, you'll find source code for GridWorld's BoxBug class and its driver, BoxBugRunner. The BoxBug class extends the Bug class. BoxBug's constructor accepts a length value, and its overriding act method causes BoxBug objects to turn 90 degrees clockwise after length steps in any one direction. BoxBugs with length < 9 eventually run around square boundaries, some of which are inside the grid's boundaries. It's not hard to imagine how you might modify

BoxBug and extend Bug differently for the CircleBug and SpiralBug classes.5

The Critter Class

GridWorld's framework/info/gridworld/actor directory also contains code for a Critter class, which extends the Actor class. You can drive Critter from CritterRunner, which is in GridWorld's projects/critters directory. Critter's extension of Actor has greater versatility than Bug's extension of Actor. Instead of specifying detailed behavior within its act method, the Critter class delegates detailed specification to five subordinate methods, like this:

```
/**
* A critter acts by getting a list of other actors, processing
* that list, getting locations to move to, selecting one of them,
* and moving to the selected location.
*/
@Override
public void act()
{
  if (getGrid() == null)
  {
    return;
  }
  ArrayList<Actor> actors = getActors();
  processActors(actors);
  ArrayList<Location> moveLocs = getMoveLocations();
  Location loc = selectMoveLocation(moveLocs);
  makeMove(loc);
} // end act
```

In GridWorld’s Critter class, each of the five delegate methods is very simple. The getActors method returns all immediate neighbors—actors in adjacent squares. The processActors method removes all parameter elements that are not of type Rock or of type Critter—eats neighbors who are edible and different. The moveLocs method returns all empty adjacent locations. The selectMoveLocation randomly selects one of its parameter elements—a valid empty adjacent location. If there is none, it returns the current location. The selectMoveLocation moves to the parameter location—the previously selected location.

You can see Critter behavior by executing the CritterRunner class in the GridWorld directory, projects/critters. Each of the five delegate methods in the Critter class does something reasonable. However, if you read the code's comments, you'll see that the designer expects us to extend the Critter class and override one or more of these five methods. They are *default* methods.

GridWorld Extensions of the Critter Class

GridWorld's first extension of the Critter class is its ChameleonCritter class. This class overrides just two of the five delegate methods, processActors and makeMove. In its processActors method, instead of eating its neighbors, a ChameleonCritter changes its color to match the color of a neighbor or one of several neighbors selected at random. In its makeMove method, a ChameleonCritter turns toward the location it will move to before calling super.makeMove to actually make the move. You can see how this works by executing ChameleonRunner, which is also in the projects/critters directory.

GridWorld's second extension of the Critter class is its CrabCritter class. This class and its driver, CrabRunner, are also in the projects/critters directory. CrabCritter differs from Critter in that it eats only those neighbors who are directly ahead or on either side of the position directly ahead. It differs also in that it can only move sideways, either left or right. If both side locations are available, it randomly chooses one of them. If neither is available, it turns by 90 degrees, randomly either left or right.

CrabCritter2 Extension of the Critter Class

Page 622

Looking at either BugRunner or CrabRunner output, you'll notice that eventually bugs tend to run clockwise around the grid's boundary. In contrast, CrabRunner's crabs tend to jump back and forth in the middle. Put yourself in the crabs' shells. Suppose you want to catch and eat bugs and flowers more efficiently than a CrabCritter. Maybe it would be better to hunt along the boundary and move around that boundary in the counterclockwise direction. Then bugs would be running right into your waiting claws.

These considerations motivate us to create a different breed of crab. Figure 13.23a contains the beginning of an alternate extension of Critter called

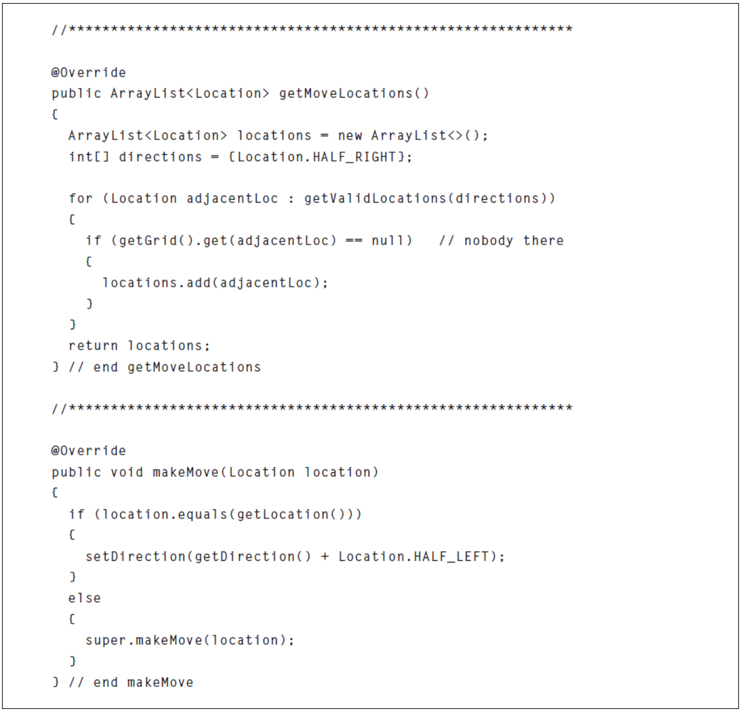
CrabCritter2. CrabCritter2 uses the same imports and has the same constructor as the original CrabCritter. In Figure 13.23a the import of java.awt.Color is from Java's legacy GUI library.[6] CrabCritter2 uses java.awt.Color in the setColor method which our CrabCritter2 class inherits from the Actor class to change Actor's default[7] BLUE coloring to RED. The getActors method is essentially unchanged, but we make a few cosmetic updates. For example, we change the name of CrabCritter's getLocationsInDirections method to getValidLocations.

The next part of CrabCritter2 appears in Figure 13.23b. This contains the other two overriding methods, getMoveLocations and makeMove. CrabCritter2's getMoveLocations method differs from CrabCritter's in that CrabCritter2 can move only diagonally forward and to the right;

that is, +45° clockwise from straight ahead. Consequently, the directions array always contains exactly one direction. If there is a valid location in that direction, the getMoveLocations method returns an ArrayList with that one location. If there is not a valid location in that direction, the getMoveLocations method returns an empty ArrayList.

```
/***************************************************************
* CrabCritter2.java
* Dean & Dean
*
* This modifies GridWorld crabs so that they turn half-left.
* GridWorld is distributed under the terms of the GNU General
* Public License as published by the Free Software Foundation.
***************************************************************/

import info.gridworld.actor.*;   // Actor, Critter
import info.gridworld.grid.*;    // Location, Grid
import java.awt.Color;
import java.util.ArrayList;

public class CrabCritter2 extends Critter
{
  public CrabCritter2()
  {
    setColor(Color.RED);
  } // end constructor

  //************************************************************

  @Override
  public ArrayList<Actor> getActors()
  {
    ArrayList<Actor> actors = new ArrayList<>();
    int[] directions =
      {Location.AHEAD, Location.HALF_LEFT, Location.HALF_RIGHT};
    Actor actor;

    for (Location adjacentLoc : getValidLocations(directions))
    {
      actor = getGrid().get(adjacentLoc);
      if (actor != null)
      {
        actors.add(actor);
      }
    }
    return actors;
  } // end getActors
```

Figure 13.23a CrabCritter2—part A



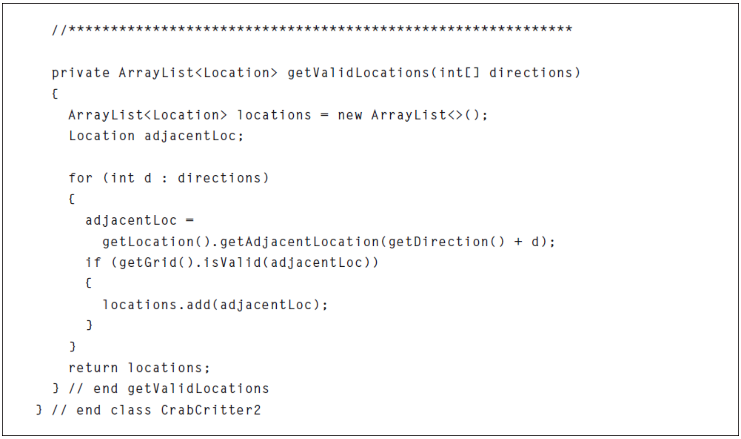
```
//*************************************************************

@Override
public ArrayList<Location> getMoveLocations()
{
  ArrayList<Location> locations = new ArrayList<>();
  int[] directions = {Location.HALF_RIGHT};

  for (Location adjacentLoc : getValidLocations(directions))
  {
    if (getGrid().get(adjacentLoc) == null)   // nobody there
    {
      locations.add(adjacentLoc);
    }
  }
  return locations;
} // end getMoveLocations

//*************************************************************

@Override
public void makeMove(Location location)
{
  if (location.equals(getLocation()))
  {
    setDirection(getDirection() + Location.HALF_LEFT);
  }
  else
  {
    super.makeMove(location);
  }
} // end makeMove
```

Figure 13.23b CrabCritter2—part B

Page 624

If the ArrayList returned by ge3tMoveLocations is empty, Criter's selectMoveLocation returns the current location. Otherwise, selectMoveLocation returns the location in the ArrayList. CrabCritter2's makeMove method takes this location as its argument. It differs from CrabCritter's makeMove method as follows: If a CrabCritter2 needs to turn, it always turns exactly the same way, Location.HALF_LEFT or −45°. This makes CrabCritter2 crabs eventually move around the grid boundary like bugs, but in the opposite direction.

Figure 13.23c shows the final method in our CrabCritter2 class. It is essentially the same as the CrabCritter method called getLocationsInDirections, but with a slightly different style. Although we created our CrabCritter2 class to implement significantly different behavior than GridWorld's CrabCritter implements, we got most of the algorithm and a substantial amount of the code from GridWorld's implementation. This example shows you another way to reuse existing software.



```
    //**************************************************************

    private ArrayList<Location> getValidLocations(int[] directions)
    {
      ArrayList<Location> locations = new ArrayList<>();
      Location adjacentLoc;

      for (int d : directions)
      {
        adjacentLoc =
          getLocation().getAdjacentLocation(getDirection() + d);
        if (getGrid().isValid(adjacentLoc))
        {
          locations.add(adjacentLoc);
        }
      }
      return locations;
    } // end getValidLocations
  } // end class CrabCritter2
```

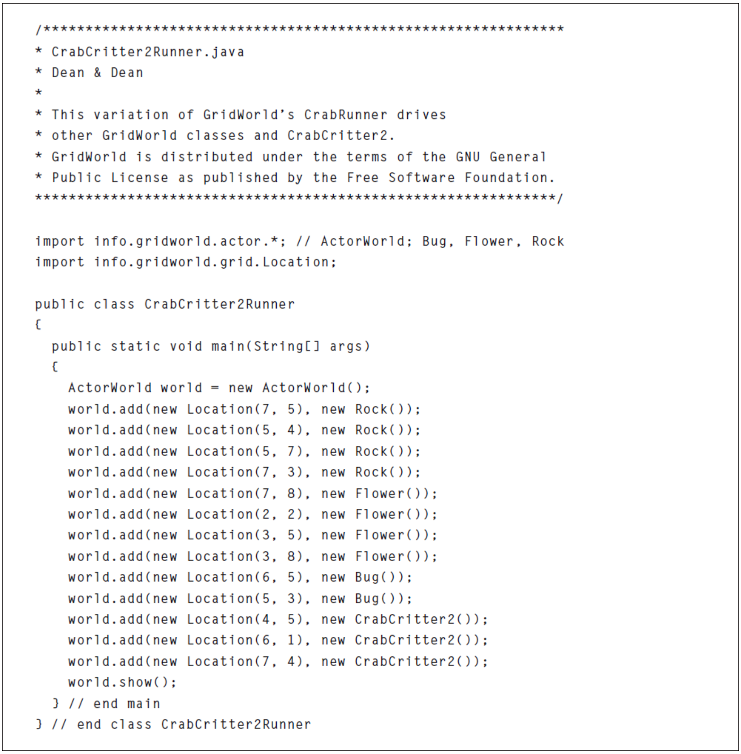
Figure 13.23c CrabCritter2—part C

Now look at the CrabCritter2Runner class in Figure 13.24. This is the driver for our CrabCritter2 program. We intentionally made it as similar as possible to GridWorld's CrabRunner class, to facilitate comparison. The only difference is that we substituted CrabCritter2 objects for CrabCritter objects.

Do you recall the discussion of GridWorld images at the end of the last chapter? If we do nothing with GridWorld's images, when we run our CrabCritter2 program, the JVM looks for a CrabCritter2.gif file in the CrabCritter2 directory. Failing that, it goes one step up the inheritance hierarchy to the Critter class, which is in the framework/info/gridworld/actor directory. Figure 13.22a shows that this directory contains a Critter.gif file, so the program will use that file's image, which looks something like a squirrel. But we want our image to look like the CrabCritter.gif image in Figure 13.22b. So we make a copy of the CribCritter.gif image in the projects/critters directory and paste it into our CrabCritter2 directory. Then we change the name of that copied image to CrabCritter2.gif.

When we run GridWorld's original CrabCritter program, the three crabs in the middle eventually eat two of the three initial bugs. But one bug survives, continuously eating and replanting flowers around the grid boundary. So the four species in

CrabCritter’s world—bugs, flowers, crabs, and rocks—all seem to survive. On the other hand, when we run our improved CrabCritter2 program, with its more efficient crabs, eventually these superior crabs consume all the bugs and all the flowers, leaving only crabs and rocks. So we achieved our goal. Hurrah! Was it a worthwhile goal —developing a predator that destroys all its own food? That question lies beyond the scope of computer science.



```
/*************************************************************
* CrabCritter2Runner.java
* Dean & Dean
*
* This variation of GridWorld's CrabRunner drives
* other GridWorld classes and CrabCritter2.
* GridWorld is distributed under the terms of the GNU General
* Public License as published by the Free Software Foundation.
*************************************************************/

import info.gridworld.actor.*; // ActorWorld; Bug, Flower, Rock
import info.gridworld.grid.Location;

public class CrabCritter2Runner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.add(new Location(7, 5), new Rock());
    world.add(new Location(5, 4), new Rock());
    world.add(new Location(5, 7), new Rock());
    world.add(new Location(7, 3), new Rock());
    world.add(new Location(7, 8), new Flower());
    world.add(new Location(2, 2), new Flower());
    world.add(new Location(3, 5), new Flower());
    world.add(new Location(3, 8), new Flower());
    world.add(new Location(6, 5), new Bug());
    world.add(new Location(5, 3), new Bug());
    world.add(new Location(4, 5), new CrabCritter2());
    world.add(new Location(6, 1), new CrabCritter2());
    world.add(new Location(7, 4), new CrabCritter2());
    world.show();
  } // end main
} // end class CrabCritter2Runner
```

Figure 13.24 CrabCritter2Runner—the driver for the CrabCritter2 class in Figures 13.23a, 13.23b, and 13.23c

13.12 Problem Solving with Association Classes (Optional)

Aggregation, composition, and

inheritance implement some of the most common kinds of associations among classes and objects—a has-a association for aggregation and composition, and an is-a association for inheritance. Be aware that there are many other possible kinds of associations, which you can conjure up easily by rattling off a few verb phrases, like “be next to. . . ,” “get. . .from. . . ,” “set. . .in. . . ,” “make. . . with. . . ,” run. . .toward. . . ,” “sell. . .to. . . ,” and so on. Typically, these other kinds of associations are more complicated than is-a or has-a associations. This section describes a powerful way to model other associations.

As you have seen, you can implement simple aggregation and composition associations by giving the container object a reference to each component object. This reference allows container object code to invoke component object methods. But for other kinds of associations, you may need multiple references and additional variables and methods. In other words, you may need a separate class just to describe the association. Such a class is called an *association class*. An association class defines an association object that represents a relationship among other objects. An association object is like an aggregation/composition container, in that it has instance variables that refer to other objects. But it's different in that the objects it refers to also refer to it, and each cannot contain the other. An association object typically receives references to the objects it associates when it is constructed. Whereas an aggregation/composition container contains its component objects, an association object just "knows about" the objects it associates.

Now let's see how this might apply to our previous Dealership program. What we've done so far with that program isn't much to brag about. We created a company with a sales manager, some salespeople, and some cars. What about customers? What about sales? Suppose we add a Customer class to our Dealership program. Then suppose some eager salesperson finally makes a sale to that first customer. The next question is, where do we put the information about that sale? In the Dealership class? In the SalesPerson class (as we seem to be doing in Figure 13.6)? In the Car class? In the Customer class? Technically, we could put that information in any one of these classes, and then put references to that class in whatever classes need access to that information. We could also scatter the information around among the participating classes in some way. No matter which of these alternatives we picked, however, from some points of view, what we did would seem inappropriate.

A more elegant solution is to encapsulate all of the sale information into one association class and give that class a name that describes the association. That's what we portray in Figure 13.25, which shows an abbreviated class diagram of another version of our previous Dealership program. First, look at the Customer class. Because a customer is a person, just like the sales manager and salespeople, we can use inheritance to reduce code and avoid redundancy in the Customer class by making the Customer class extend the Person class. Second, look at the Sale class. The Sale class appears as just another component in the Dealership class diagram. The one-to-many multiplicity suggests that its objects are elements of an ArrayList, perhaps named sales, which is instantiated in an enhanced version of the Dealership constructor. As far as the dealership is concerned, a Sale would be just another type of aggregation or composition component, like a SalesPerson2 or a Car.

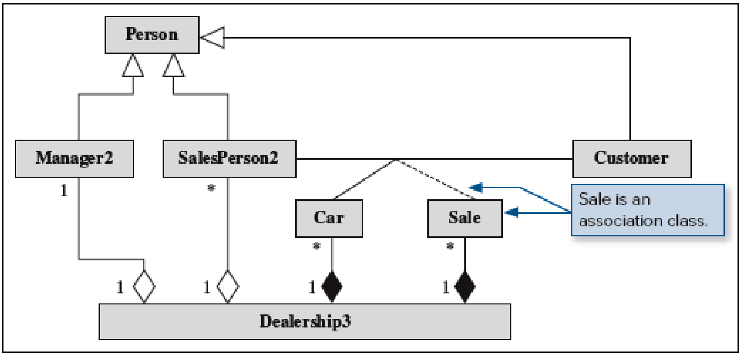
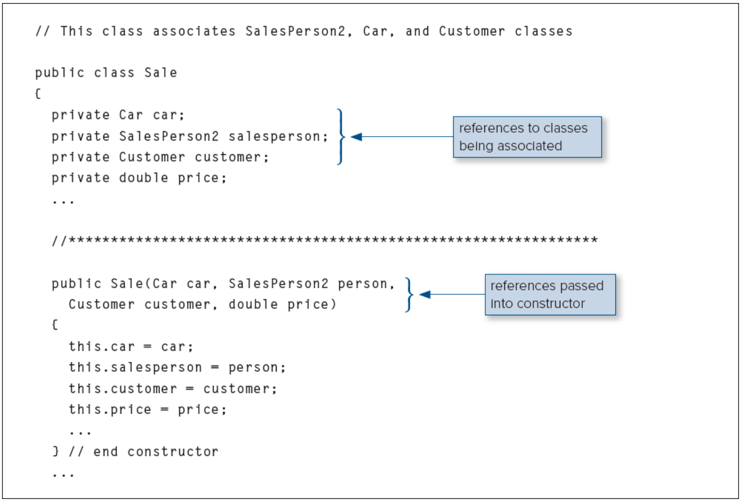

Person
Manager2
SalesPerson2
Customer
Car
Sale
Sale is an association class.
1
*
*
*
1
1
1
1
Dealership3

Figure 13.25 Class diagram for another car dealership program with customers and a salesperson-car-customer association

However, a sale is not a physical entity, like an organism or car is. It's a process—or event—that associates a group of entities. So the Sale class needs to be an association class. What types of objects participate in a Sale association? There’s a Car, there’s a SalesPerson2, and there’s a Customer. Notice how the UML class diagram in Figure 13.25 uses simple solid association lines to interconnect all normal classes that participate in an association. The UML standard suggests ways to decorate these association lines with additional symbols and nomenclature, but just the lines shown convey the message—the idea of an association among objects of the Car, SalesPerson2, and Customer classes. The dashed line that connects the Sale class to the solid association lines graphically identifies the Sale class as an association class describing the related association. The code fragment in Figure 13.26 illustrates the Sale constructor.



```
// This class associates SalesPerson2, Car, and Customer classes

public class Sale
{
  private Car car;
  private SalesPerson2 salesperson;
  private Customer customer;
  private double price;
  ...

  //*****************************************************************

  public Sale(Car car, SalesPerson2 person,
    Customer customer, double price)
  {
    this.car = car;
    this.salesperson = person;
    this.customer = customer;
    this.price = price;
    ...
  } // end constructor
  ...
```

references to classes being associated
references passed into constructor

Figure 13.26 Partial implementation of the Sale class shown in Figure 13.25

Caveat—Don’t Try to Inherit from an Association Participant

You might be tempted to try to use inheritance to create an association class because you might think that would give you “free access” to at least one of the participants in the association. Don’t try to do that. All you’d get would be the ability to make an enhanced clone of one of the objects you want to associate, and you’d have to copy all the details between the clone and the real thing—a waste of effort. Treat an association like an aggregation, with references to the participating objects passed into the association constructor.

Summary

- Object-oriented languages help you organize things and concepts into two basic kinds of hierarchies—a has-a hierarchy for components in an aggregation or composition, and an is-a hierarchy for types in an inheritance.
- An aggregation or composition hierarchy exists when one large object contains several smaller (component) objects.
- For a given whole-part class relationship, if the container contains the only reference to a

component, the component association is composition. Otherwise, it’s aggregation.

- In an inheritance hierarchy, subclasses inherit all the variables and methods of the superclasses above them, and they typically add more variables and methods to what they inherit.

- To minimize descriptive duplication, organize your ideas so that only the concepts at the very bottom of an inheritance hierarchy (the leaves of the upside-down tree) are specific enough to represent real objects.

- To enable class B to inherit all the variables and methods in class A and all of class A’s ancestors, append extends A to the end of class B’s heading.

- A constructor should initialize the variables it inherits by immediately calling its superclass's constructor with the statement: super(*arguments*);

- You can override an inherited method by writing a different version of the inherited method in the derived class. Overriding occurs automatically if you use the same method name and the same - sequence of parameter types, but if you do this, you must also use the same return type.
- You can access an overridden method by prefixing the common method name with super and then a dot.
- A final access modifier on a method keeps that method from being overridden. A final access modifier on a class keeps that class from being extended.
- Programmers frequently use combinations of aggregation,

composition, and inheritance to deal with different aspects of an overall programming problem. In a UML class diagram, both relationships are represented by solid lines between related classes, and these lines are called associations. In a composition/ aggregation association, there is a solid/hollow diamond at the container end of each association line. In a hierarchical association, there is a hollow arrowhead at the superclass end of the association line.

- Inheritance allows you to reuse code that was written for another context.

- When you have a complicated association among objects, it may help to gather references to those - objects together into a common association class.

Review Questions

§13.2 Composition and Aggregation

1. In a UML diagram, what does an asterisk (*) indicate?

2. In a UML diagram, what does a solid diamond indicate?

§13.3 Inheritance Overview

3. Explain how using an inheritance hierarchy can lead to code reusability.

4. What are two synonyms for a superclass?

5. What are two synonyms for a subclass?

Page 630

§13.4 Implementation of Person/ Employee/FullTime Hierarchy

6. How do you tell the compiler

that a particular class is derived from another class?

7. Based on the UML diagram in Figure 13.9, an instance of the PartTime class includes the following instance variables: name and id. (T / F)

§13.5 Constructors in a Subclass

8. In a subclass's constructor, what do you have to do if you want to begin the constructor with a call to the superclass's zero-parameter constructor?

§13.6 Method Overriding

9. If a superclass and a subclass define methods having the same name and the same sequence of parameter types, and an object of the subclass calls the method without specifying which version, Java generates a runtime error. (T / F).

10. If a subclass method overrides a method in the superclass, is it still possible to call the method in the superclass from the subclass?

11. If a superclass declares a variable to be private, can you access it directly from a subclass?

§13.7 Using the Person/Employee/FullTime Hierarchy

12. If you wish to call a superclass method, you must always prefix the method name with super. (T / F)

§13.8 The final Access Modifier

13. A final method is called “final” because it’s allowed to contain only named constants, not regular variables. (T / F)

§13.9 Using Inheritance with Aggregation and Composition

14. Composition and inheritance are alternative programming techniques for representing what is essentially the same kind of real-world relationship. (T / F).

§13.10 Design Practice with Card Game Example

15. A Deck is a group of cards and a Deck has a group of cards. In our example, it's better to choose the is-a relationship and implement inheritance. In this case, why is inheritance a better choice than composition?

§13.11 Problem Solving with Association Classes (Optional)

16. It's possible to support an association with references, variables, and methods in existing classes. What's the advantage of using an association class instead?

Exercises

1. [after §13.2] This exercise should be used in combination with Exercises 2 and 3. Implement a Point class with the following features. Declare a position instance variable—an array of doubles that holds the point's x, y, and z coordinate values. Provide a constructor that receives x, y, and z values and instantiates the position array with those values. Provide a shift method with an offset parameter (an array that holds x, y, and z values) that adjusts the point's position in the directions specified by the passed-in offset. Implement the shift method so that it supports chaining. Provide an accessor method that returns the point's position instance variable.

2. [after §13.2] This exercise

should be used in conjunction with Exercises 1 and 3. Implement a Sphere class with the following features. Declare two instance variables—center (whose type is Point) and radius (whose type is double). Provide a constructor that receives center and radius values and uses them to initialize the two instance variables. Provide a shift method with an offset parameter (an array that holds x, y, and z values) that adjusts the sphere's position by shifting the sphere's center point in the directions specified by the passed-in offset. Provide a scale method with a factor parameter (a double) that adjusts the sphere's radius by multiplying it by the passed-in factor. Implement the shift and scale methods so they support chaining. Provide a describe method that displays (1) the x, y, and z coordinates for the sphere's center point and (2) the sphere's radius.

3. [after §13.2] This exercise should be used in conjunction with Exercises 1 and 2. Implement a SphereDriver class with a main method that performs the following tasks. Instantiate a Point at x = 10, y = 15, and z = 20. Instantiate a Sphere centered at that point and having a radius of 8. Call that sphere's describe method. Shift the sphere's location by +3.0 in the x-direction, −2.5 in the y-direction, and +4.0 in the z-direction. Increase its radius by a factor of 1.5. Then call its describe method again.

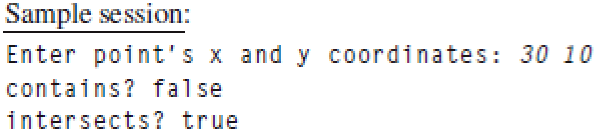
4. [after §13.3] Suppose that you have three classes—HomeInsurance (insurance for a home), InsurancePolicy (insurance for anything), and FloodDamageInsurance (insurance for flood damage to a home). Assume that the three classes form an inheritance hierarchy. Describe the inheritance relationships that exist between classes. For each relationship, specify the superclass and subclass.

5. [after §13.3] Suppose you want to design a program that handles various renewable energy sources. Implement your design as a UML diagram with 4 classes (RenewableEnergy, Hydro, Wind, and Solar) and 10 variables (initialCapitalInvestment, peakPowerOutput, averagePowerOutput, heightDifference, peakFlow, averageFlow, maxWindSpeed, averageWindSpeed, latitude, and clearnessFactor) that are declared as instance variables in the classes. Draw a UML class diagram with class names, variable names, and inheritance arrows. Assume all variables are private. You may omit type specifications, constructors, and methods.

6. [after §13.4] CircleProperties program:

Java's API classes make extensive use of inheritance. For example, Oracle's Java API documentation shows that the javafx.scene.shape package has a class named Circle that is a child of a class named Shape, which is itself a child of a class named Node. Circle inherits more than 30 methods from its parent, Shape, and more than 330 methods from its grandparent, Node. One of Circle's constructors has three double parameters, the first two for the x and y coordinates of its center and the third for its radius.

Implement a program that creates a circle centered at 50, 30 with a radius of 20. Have the program ask the user to supply an additional pair of x and y coordinate values. Use Node’s contains method to see if the point corresponding to these (other) user-entered x and y coordinates is within the circle. Also use Node’s intersects method to see if there is any overlap between the circle and a rectangle whose upper left corner is at the user-entered x and y coordinates and whose width and height are 20 and 15, respectively.



Sample session:

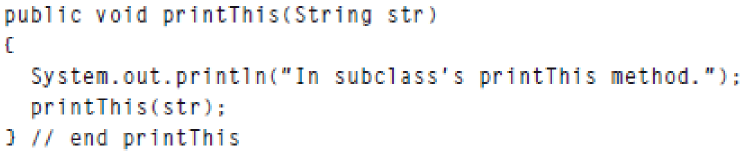
```
Enter point's x and y coordinates: 30 10
contains? false
intersects? true
```

7. [after §13.4] EllipseDriver program.

Compile and execute the following program. Provide the program’s output as part of your answer. Using Oracle’s or OpenJFX’s Java documentation as a resource, figure out what the program does and how it works. Insert comments at indicated places in the program’s code that describe what the program does in general and what various blocks of code do in particular. Note: The extended Application class provides a main method which this class inherits, and that inherited main method calls this class’s overriding start method instead of the start method in the Application class.

```
/*******************************************************
* EllipseDriver.java
* Dean & Dean
*
* <comment describing overall function>
*******************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;                  // Group & Scene
import javafx.scene.shape.Ellipse;
import javafx.scene.paint.Color;

public class EllipseDriver extends Application
{
  public void start(Stage stage)
  {
    // <comment describing constructor parameters>
    Ellipse ellipse = new Ellipse(50, 30, 20, 15);

    // <comment describing what next two statements do>
    Group root = new Group(ellipse);
    Scene scene = new Scene(root, 600, 300);

    // <comment describing what next three statements do>
    ellipse.setFill(Color.LIGHTGRAY);
    ellipse.setStroke(Color.BLACK);
    ellipse.setStrokeWidth(3);

    // <comment describing what next three statements do>
    stage.setScene(scene);
    stage.setTitle("Ellipse");
    stage.show();
  }
} // end EllipseDriver class
```

8. [after §13.6] Suppose you have two classes related by inheritance that both contain a one-parameter method named printThis. Here’s the subclass’s version of printThis:



```
public void printThis(String str)
{
  System.out.println("In subclass's printThis method.");
  printThis(str);
} // end printThis
```

a) What problem occurs when the above printThis method executes?

b) How can you fix the problem?

9. [after §13.8] What does it mean when you use the final modifier for a class?

10. [after §13.8] What does it mean when you use the final modifier for a method?

Page 633

11. [after §13.9] Fill in the blanks:

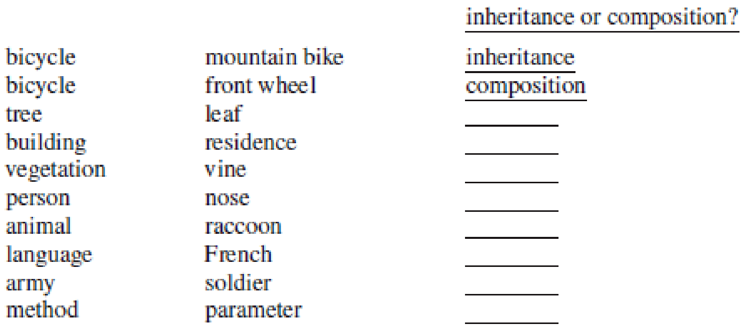
If a Y class “is a” special form of an X class, there is a(n) __________ association, and the right side of Y’s class heading will contain the words __________.

If an object X “has an” object Y

and also "has an" object Z, there is a(n) _________ association, and X's class definition will contain declarations for _________ variables.

12. [after §13.9] Identification of type of association:

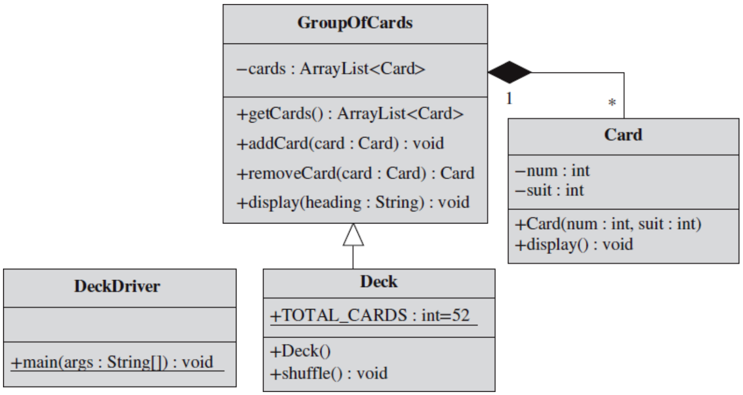
Given the following list of word pairs, for each word pair, identify the association between the two words. That is, identify whether the two words are related by composition or inheritance. To get you started, we've provided the answers to the first two word pairs. Vehicle and spare are related by composition because a vehicle "has a" spare (tire or tube). Vehicle and bicycle are related by inheritance because a bicycle "is a" vehicle.



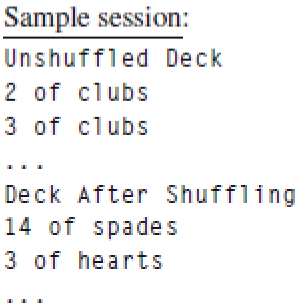
		inheritance or composition?
bicycle	mountain bike	inheritance
bicycle	front wheel	composition
tree	leaf	
building	residence	
vegetation	vine	
person	nose	
animal	raccoon	
language	French	
army	soldier	
method	parameter	

13. [after §13.10] Shuffling:

Suppose you are developing a card game program like that suggested earlier in the chapter, but you want to use an ArrayList instead of an array. The following UML class diagram shows your plan:


GroupOfCards
−cards : ArrayList<Card>
+getCards() : ArrayList<Card>
+addCard(card : Card) : void
+removeCard(card : Card) : Card
+display(heading : String) : void
1
*
Card
−num : int
−suit : int
+Card(num : int, suit : int)
+display() : void
DeckDriver
+main(args : String[]) : void
Deck
+TOTAL_CARDS : int=52
+Deck()
+shuffle() : void

Assume you have implemented the Card class and the GroupOfCards class. (The next exercise will ask you to do this.) Assume that the addCard method adds its passed-in card to the end of the ArrayList. Assume that the removeCard method deletes the passed-in card from the ArrayList (reducing the ArrayList's size by one) and returns a reference to the removed card. Provide Java code for the Deck and DeckDriver classes. In your Deck class, you must implement a shuffle method that relies on the addCard and removeCard methods. You are not allowed to use the Collection class's shuffle method. When you execute the driver, you should get output like that in the following sample session, with randomness altering the output after shuffling:



Sample session:

```
Unshuffled Deck
2 of clubs
3 of clubs
...
Deck After Shuffling
14 of spades
3 of hearts
...
```

Hint: To shuffle the deck, in the Deck class's shuffle method, use a for loop that starts with unshuffled = getCards().size() and steps down to 1. In each iteration, use Math.random to pick an index in the unshuffled range, remove the card at that index, and then add it back to the ArrayList.

14. [after §13.10] This is a continuation of the card game program described in the prior exercise. Implement the Card and GroupOfCards classes.

15. [after §13.2] (Case Study) GridWorld's firstProject file contains the BugRunner driver:

```
public class BugRunner
{
  public static void main(String[] args)
  {
    ActorWorld world = new ActorWorld();
    world.add(new Bug());
    world.add(new Rock());
    world.show();
  } // end main
} // end class BugRunner
```

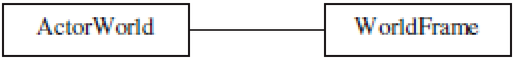
> BugRunner starts by creating an instance of ActorWorld named world. ActorWorld's constructor immediately creates a 10 × 10 BoundedGrid. Next, BugRunner asks the ActorWorld object to add a new Bug at a random location, and it then adds a new Rock at a different random location. ActorWorld's add method tells each new occupant where it is supposed to go in the grid and then tells that occupant to put itself into the grid at that location.

BugRunner's last statement calls ActorWorld's show method. This show method instantiates a WorldFrame (a

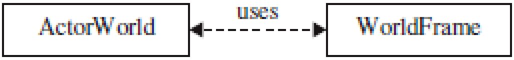
GUI window) with an argument referring back to the ActorWorld object.

WorldFrame’s constructor instantiates a GridPanel. Then it uses its reference to the ActorWorld object to get a reference to the BoundedGrid, and it passes this BoundedGrid reference on to the new GridPanel. GridPanel’s drawOccupants method uses this BoundedGrid reference to call BoundedGrid’s getOccupiedLocations method. This allows GridPanel’s paintComponent method to put bug and rock images in their proper places each time it repaints the screen.

Provide an abbreviated UML class diagram that contains all classes identified in the above two paragraphs, except for the BugRunner class. Use appropriate composition (filled diamond) and aggregation (hollow diamond) links. Include multiplicity symbols where one of the multiplicities might be greater than 1. For the relationship between ActorWorld and WorldFrame, use a simple bidirectional association, like this:



Or use a bidirectional dependence, like this:

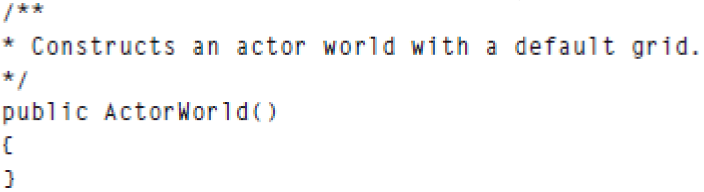


16. [after §13.4] (Case Study) Using GridWorld documentation, provide a UML class diagram for the Actor class and its descendants, Bug, Rock, and Flower. This should be similar to this chapter's UML class diagram for the Person class and its descendants. Include UML specifications of all class and instance variables and constants, as well as all methods in these classes.

17. [after §13.5] (Case Study) This chapter's Exercise 15 presents code containing the statement

> ActorWorld world = new ActorWorld();

After that, we said, "ActorWorld's constructor immediately creates a 10 × 10 BoundedGrid." But if you look at GridWorld's source code for the ActorWorld class, you'll see that the zero-parameter constructor is empty:



```
/**
* Constructs an actor world with a default grid.
*/
public ActorWorld()
{
}
```

How does this empty constructor create "a 10 × 10 BoundedGrid"?

18. [after §13.6] (Case Study) Find and describe the explicit calls to superclass members in the ActorWorld, BoundedGrid, Bug, Rock, WorldFrame, and GridPanel classes, mentioned in Exercise 15, above. In each class that makes such a call, identify the class, the method or constructor that includes the call, the constructor or method that's called, and the class in which that called member resides.

19. [after §13.6] (Case Study) Create a RandomBug class that extends the Bug class in the GridWorld program. In this extended class, override the move method inherited from Bug. Do this by copying the code from Bug's move method and then replacing the code fragment that creates flowers and deposits them in the bug's wake with a setDirection call that randomly selects one of the directions, 0, 45, 90, 135, 180, 225, 270, or 315. Get a random value the same way that GridWorld gets a random value for a bounded grid in the World class's getRandomEmptyLocation method. But instead of using generator.nextInt(emptyLocs.size()), use 45 * generator.nextInt(8).

Implement a RandomBugRunner class to drive your new RandomBug class. In this class's main method, instantiate an ActorWorld object named world. Then have world add ten RandomBug objects, and then have world call its show method.

Executing RandomBugRunner should generate a display similar to that in Figure 11.15, except this time there will be ten randomly located bugs and no rock. Push the Step button a few times to see how the bugs move. Then click World > UnboundedGrid, adjust the sliders on the sides to move the cluster of bugs to the center of the larger display, click Run, and watch the bugs gradually spread apart as time passes.

Review Question Solutions

---

1. A * on a UML diagram means the multiplicity can be “any number.”

2. In a UML diagram, a solid diamond is placed on an association line next to a containing class in a composition association. It indicates that the containing class exclusively contains the class that’s at the other end of the association line.

3. Putting common code from two classes into a superclass is one example of code reusability. Code reusability can also take place when you want to add a significant chunk of functionality to an existing class, and you implement the solution with a new subclass.

4. Two synonyms for a superclass—parent class, base class.

5. Two synonyms for a subclass—child class, derived class.

6. To tell the compiler that a class is derived from another class, you write extends *other-class* at the end of your new class's heading line.

7. True. An instance of a subclass includes that class's instance variables and its ancestors' instance variables.

8. Nothing. This happens automatically. You can, however, preempt this by writing super(); as the first line in your derived constructor.

9. False. There is no problem. The JVM selects the method in the subclass.

10. Yes. In the call statement, preface the common method name with super.

11. No. If a superclass's instance variable is private, you cannot access it directly from a subclass. You can access it by calling an accessor method (assuming the accessor is public). In calling the superclass's method, there's no need to prefix the method call with a reference dot.

12. False. The super. prefix is necessary only when you want to call a superclass method that has been overridden.

13. False. A final method is allowed to contain regular variables. It's called "final" because it's illegal to create an overriding version of the method in a subclass.

14. False. Composition and inheritance are completely different class relationships. Composition is when a class is comprised of nontrivial constituent parts and the parts are defined to be classes. Inheritance is when one class is a more detailed version of another class. More formally, inheritance is when one class, a subclass, inherits variables and methods from another class, a superclass.

15. Inheritance is better because with this example, there’s a second class that is also a group of cards. Because there are two classes that share some of the same properties, you should put those common properties in a shared superclass, GroupOfCards. Doing this promotes software reuse and avoids code redundancy.

16. You can make a complicated association easier to recognize and understand by organizing the references to all association participants and other association information and methods in a single class that represents the association only.

---

[1] Physical ailments can also be inherited. At age 30, author Ray tore his left Achilles tendon. Sons Stan and John did the same when they turned 30.

[2] This sequence is the same as the sequence that occurs naturally in the embryonic development of a living creature. The characteristics that develop first are the most ancient ones.

[3] The BugRunner class is in the GridWorld.jar file in the projects/firstProject directory. The other classes used by this program are in the same

.jar file in the framework/info/gridworld directory in the subdirectories: actor, grid, gui, and world.

[4] See Section 5.9 in Chapter 5 for more information about image formats.

[5] These are two of the exercises in GridWorld's "Student Manual," written by Chris Nevison, Barbara Cloud Wells, and Chris Renard, https://secure-media.collegeboard.org/apc/GridWorld_Case_Study_Student_Manual_with_

[6] The java.awt.Color class differs from the javafx.scene.paint.Color class and won't work in JavaFX programs.

[7] As described at the end of Section 13.5, CrabCritter2's zero-parameter constructor automatically calls Actor's zero-parameter constructor, which initializes color and other properties of Actor descendants.

Inheritance and Polymorphism

Objectives

- Understand the role of the Object class.
- Learn why you need to redefine the equals and toString methods.
- Learn how polymorphism and

dynamic binding improve program versatility.

- Understand what the compiler checks and what the JVM does when a reference variable is associated with a method name.

- Understand the constraints affecting assignment of an object of one class to a reference variable of another class.

- See how to use an array of ancestor reference variables to implement polymorphism among descendant methods.

- See how an abstract method declaration in an abstract superclass eliminates the need for a dummy method definition in the superclass.

- See how you can use an interface to specify common method headings, store common constants, and implement multiple inheritance.

- Learn where to use protected member access.
- Optionally, learn how to draw a three-dimensional object.

Outline

14.1 Introduction 638

14.2 The Object Class and Automatic Type Promotion 638

14.3 The equals Method 639

14.4 The toString Method 643

14.5 Polymorphism and Dynamic Binding 648

14.6 Assignments When the Two Sides' Classes Are Different 653

14.7 Polymorphism with

Arrays 654

14.8 abstract Methods and Classes 660

14.9 Interfaces 663

14.10 The protected Access Modifier 673

14.11 GUI Track: Three-Dimensional Graphics (Optional) 677

14.1 Introduction

This is the second of two chapters on inheritance. The previous chapter applied a broad brush stroke to fundamental inheritance concepts. In this chapter, we narrow the focus and describe several inheritance-related topics in depth. We start with the Object class, which is the (Java API) superclass of all other classes. We then discuss one of the cornerstones of object-oriented programming (OOP)—polymorphism. *Polymorphism* is the ability for a particular method call to perform different operations at different times. It occurs when you have a reference variable that refers to different types of objects during the course of a program's execution. When the reference variable calls the polymorphic method, the reference variable's object type determines which method is called at that time. Pretty cool, eh? Polymorphism provides programs with a great deal of power and versatility.

After introducing polymorphism, we describe its partner, dynamic binding. *Dynamic binding* is the mechanism used by Java to implement polymorphism. We then provide alternative implementations of polymorphism, using abstract classes to make code cleaner.

Chapter 10 introduced you to several examples of interfaces in the prewritten Java collections framework. In this chapter, we'll show you how to define your own multiple-signature interfaces and use them to manage code development, store universal constants, and make polymorphism more versatile.

Then we'll describe the protected modifier, which simplifies access to inherited code. Finally, in an optional section, we present two three-dimensional graphics examples that illustrate polymorphism with the Java API.

The material in this chapter is relatively difficult, but once you get it, you'll truly understand what OOP is about, and you'll know how to craft elegantly structured programs.

14.2 The Object Class and

Automatic Type Promotion

The Object class is the ancestor of all other classes. It is the primordial ancestor—the root of the inheritance hierarchy. Any class that explicitly extends a superclass uses extends in its definition. Whenever anyone creates a new class that does not explicitly extend some other class, the compiler automatically makes it extend the Object class. Therefore, all classes eventually descend from the Object class. The Object class doesn't have many methods, but the ones it has are significant because they are always inherited by all other classes. In the next two sections, you'll learn about the Object class's two most important methods, equals and toString. Because any class that you write automatically includes these two methods, you need to be aware of what happens when these methods are called.

Before diving into the details of these two methods, however, we want to make you aware of a Java process that's very similar to the numerical type promotion that you studied in Chapters 3 and 12. There, you saw that in the course of making an assignment or copying an argument into a parameter, the JVM performs a promotion when the two involved types conform to a certain numerical hierarchy. For example, when an int value is assigned into a double variable, the JVM automatically promotes the int value to a double value.

An analogous promotion can also occur with reference types. When an assignment or argument passing operation involves different reference types, the JVM automatically promotes the source reference type to the target reference type if the target reference type is above the source reference type in the inheritance hierarchy. In particular, since the Object class is an ancestor of every other class, when the need arises, Java automatically promotes any class type to the Object type. The next section describes a situation that stimulates this kind of type promotion.

14.3 The equals Method

Syntax

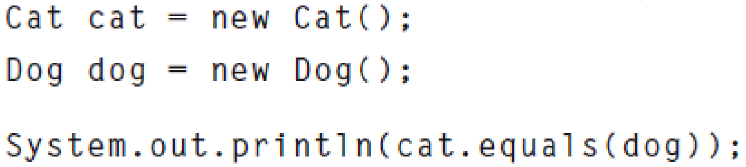
The Object class's equals method—which is inherited automatically by all other classes—has this public interface:

```
public boolean equals(Object obj)
```

Because all classes automatically inherit this method, unless a similarly defined method takes precedence, any object, objectA, can invoke this method to compare itself with any other object, objectB, with a method call like this:

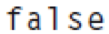
objectA.equals(objectB)

This method call returns a boolean value of either true or false. Notice that we did not specify the type of either objectA or objectB. In general, they can be instantiations of any class, and they do not need to be objects of the same class. The only constraint is that objectA must be a non-null reference. For example, if Cat and Dog classes exist, this code works correctly:



```
Cat cat = new Cat();

Dog dog = new Dog();

System.out.println(cat.equals(dog));
```

Output:



```
false
```

The equals method that is called here is the equals method that the Cat class automatically inherits from the

Object class. The parameter in this inherited method is of type Object, as specified in the method's public interface above. But the dog argument we pass to this method is not of type Object. It is of type Dog. So what's happening? When we pass the dog reference into the inherited equals method, the reference type automatically promotes from type Dog to type Object. Then the inherited equals method performs an internal test to see if the passed in dog is the same as the calling cat. Of course it is not, so the output is false, as you can see.

Semantics

Notice that we just said, "performs an internal test." Now let's focus on that mysterious "internal test." How can you tell if two objects are the same or "equal"? When you say "objectA equals objectB," you could mean this:

1. objectA is just an alias for objectB, and both objectA and objectB refer to exactly the same object.

Or you could mean this:

2. objectA and objectB are two separate objects that have the same attributes.

The equals method that all classes inherit from the Object class implements the narrowest possible meaning of the word “equals.” That is, this method returns true if and only if objectA and objectB refer to exactly the same object (definition 1). This behavior of equals is exactly the same as the behavior of the = = operator when it tests the equality of two reference variables. That operator also returns true if and only if both references refer to exactly the same object.

Suppose you have a Car class with

three instance variables, make, year, and color, and you have a constructor that initializes these instance variables with corresponding argument values. Suppose this Car class does not define an equals method itself, and the only equals method it inherits is the one it inherits automatically from the Object class. The following code illustrates that the equals method inherited from the Object class does exactly the same thing as the = = operator does:

```
Car car1 = new Car("Honda", 2014, "red");
Car car2 = car1;
Car car3 = new Car("Honda", 2014, "red");

System.out.println(car2 == car1);
System.out.println(car2.equals(car1));
System.out.println(car3 == car1);
System.out.println(car3.equals(car1));
```

different names for the same object

different objects with the same attributes

Output:

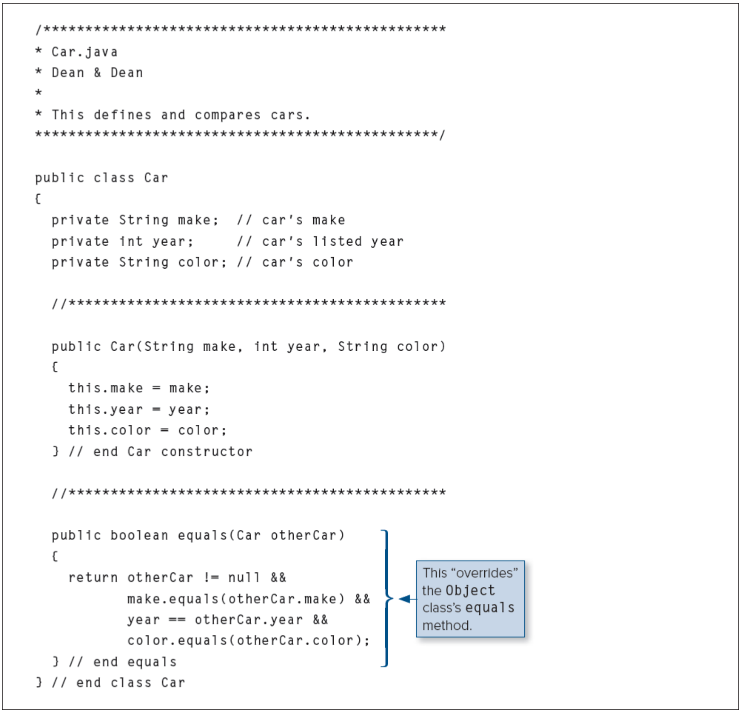
```
true
true
false
false
```

This narrow sense of the word “equals” is not always what you want. For example, suppose that your spouse decides to buy a new car, goes to a particular auto dealer, and orders a red 2014 Honda, as suggested by the above car1 instantiation. When you see the brochures that your spouse brings home, you’re impressed and decide you would like a new car for yourself too. You’d like it to be just like your spouse’s car except for the color, which you want to be blue. So you go to the same dealer and say, “I want the same car my spouse just ordered, but I want the color to be blue.” A month later, the dealer calls both you and your spouse at your separate places of work and says to each of you separately, “Your car is ready. Please come in to pick it up at 5:30 pm today.” You both show up as requested, and the dealer takes you outside and proudly exclaims, “Here it is. How do you like it?” You say “Great, it’s just what I wanted!” Then your spouse says, “But where is my car?” And the dealer replies, “But I thought you were to be joint owners of the same car, and your spouse told me to change the color of that car to blue.” Oops, somebody made a mistake. . . .

The mistake occurred in the communication between you and the dealer when you said, “the same car.” You meant the second meaning above: objectA and objectB are two separate objects which have the same attributes. But the dealer heard the first meaning above: objectA is just another name for objectB, and both objectA and objectB refer to exactly the same object.

Defining Your Own equals Method

Now let’s see how you can implement the second meaning. To do so, include in your class an explicit version of an equals method that tests for equal attributes. Then, when your program runs, and an instance of your class calls the equals method, your equals method takes precedence over Object’s equals method, and the JVM utilizes the equals method you defined. The equals method in Figure 14.1's Car class tests for equal attributes by comparing the values of all three instance variables, which are the object's attributes. It returns true only if all three instance variables have the same values, and it returns false otherwise. Notice that this equals method includes two subordinate equals method calls—one made by the make instance variable and the other made by the color instance variable. As explained in Chapter 3, these calls to String's equals method check to see if two different strings have the same character sequence.



```
/***************************************************
* Car.java
* Dean & Dean
*
* This defines and compares cars.
***************************************************/

public class Car
{
  private String make;  // car's make
  private int year;     // car's listed year
  private String color; // car's color

  //**********************************************

  public Car(String make, int year, String color)
  {
    this.make = make;
    this.year = year;
    this.color = color;
  } // end Car constructor

  //**********************************************

  public boolean equals(Car otherCar)
  {
    return otherCar != null &&
           make.equals(otherCar.make) &&
           year == otherCar.year &&
           color.equals(otherCar.color);
  } // end equals
} // end class Car
```

This "overrides" the `Object` class's `equals` method.

Figure 14.1 Car class, which defines equals to mean having the same instance variable values

In the equals method's return expression, notice the otherCar != null subexpression. If this evaluates to false (indicating that otherCar is null), Java's short-circuit evaluation keeps the computer from trying to use a null reference to access the other car's make and color reference variables.

Such short-circuit evaluation prevents runtime errors. You should always strive to make your code robust. In this case, that means you should consider the possibility of someone passing in a null value for otherCar. If null gets passed in and there's no test for null, the JVM generates a runtime error when it sees otherCar.make. This is a fairly common error—attempting to access a member from a null reference variable—and you ⚠ can avoid it easily. Just test for null prior to accessing the member. For our equals method, if otherCar is null, then the otherCar != null subexpression is false, and the return statement returns false. Returning false is appropriate because a null otherCar is clearly not the same as the calling object Car.

Get in the habit of writing equals methods for most of your programmer-defined classes. Writing equals methods is usually straightforward because they tend to look the same. Feel free to use the Car class's equals method as a template.

Remember that any reference variable can call the equals method, even if the reference variable's class doesn't define an equals method. You know what happens in that case, right? When the JVM realizes that there's no local equals method, it looks for the equals method in an ancestor class. If it doesn't find an equals method prior to reaching the Object class at the top of the tree, it uses the Object class's ⚠ equals method. This default operation often appears as a bug. To fix the bug, make sure that your classes implement their own equals methods.

As the callout in Figure 14.1 suggests, it's OK to think informally that the Car class's equals method "overrides" the Object class's equals method. But technically speaking, it is not an overriding method. If you precede your “overriding” equals method with an @Override tag, the compiler will complain and tell you it is not an overriding method. Instead, it’s a method that has the same name as a superclass method, but a different signature because the parameter’s type is different. As shown earlier, the heading of the Object class’s equals method looks like this:

```
public boolean equals(Object obj)
```

On the other hand, the heading of our Car class’s equals method looks like this:

```
public boolean equals(Car otherCar)
```

You can see that the parameter types are different—obj is an Object, while otherCar is a Car.

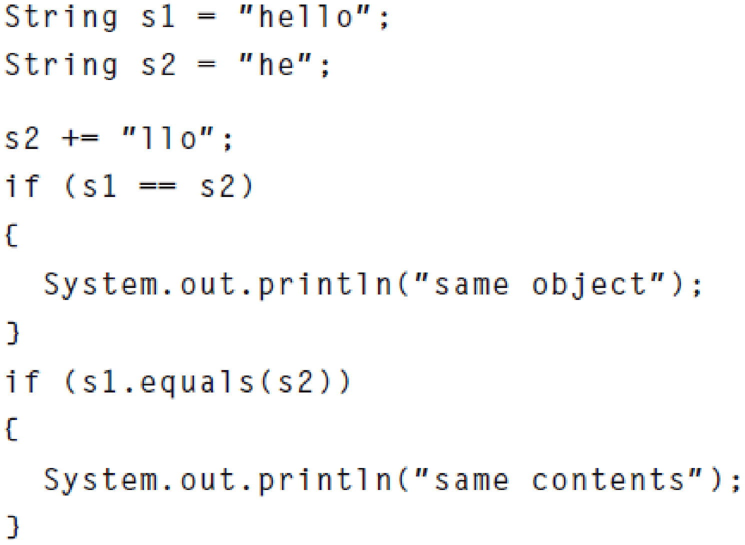
There are many situations where actually overriding a method is useful. As illustrated in the Employee and FullTime classes in Figures 13.11 and

13.12, an overriding method can use super to obtain the services of the overridden method and then provide additional statements for additional services. But there is no good reason to actually override Object's equals method because that method works just fine for equals method calls whose arguments match the generic Object parameter.

equals Methods in API Classes

Note that equals methods are built into many API classes.1 For example, the String class and the wrapper classes implement equals methods. As you'd expect, these equals methods test whether two references point to data that is identical (not whether two references point to the same object).

You've seen the String class's equals method before, so the following example should be fairly straightforward. It illustrates the difference between the = = operator and the String class's equals method. What does this code fragment print?



```
String s1 = "hello";
String s2 = "he";

s2 += "llo";
if (s1 == s2)
{
  System.out.println("same object");
}
if (s1.equals(s2))
{
  System.out.println("same contents");
}
```

This code fragment prints "same contents." Let's make sure you understand why. The = = operator returns true only if the two reference variables being compared refer to the same object. In the first if statement, s1 = = s2 returns false because s1 and s2 do not refer to the same object. In the second if statement, s1.equals(s2) returns true because the characters in the two compared strings are the same.

Actually, there's another twist to the String class. To minimize storage requirements, the Java compiler makes string references refer to the same String object whenever an assignment refers to a duplicate string literal. That's called *string pooling*. For example, suppose the preceding code included a third declaration that looked like this:

```
String s3 = "hello";
```

Then, if the first if condition were (s1 = = s3), the output would say "same object" because s1 and s3 would refer to the same "hello" string object.

14.4 The toString Method

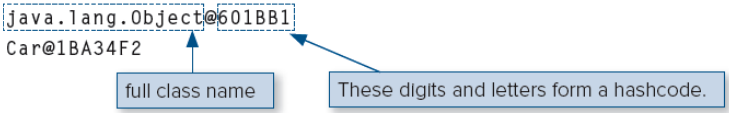
The Object Class's toString Method

Let's now consider another important

method that all classes inherit from the Object class. The Object class's toString method returns a string that's a concatenation of the calling object's full class name, an @ sign, and a sequence of digits and letters. For example, consider this code fragment:

```
Object obj = new Object();
Car car = new Car();

System.out.println(obj.toString());
System.out.println(car.toString());
```

When executed, the code fragment produces this:



```
java.lang.Object@601BB1
Car@1BA34F2
```

Note how obj.toString() generates java.lang.Object for the full class name. The full class name consists of the class name prefixed by the class's package. The Object class is in the java.lang package, so its full class name is java.lang.Object. Note how car.toString() generates Car for the full class name. Because the Car class is not part of a package, its full class name is simply Car.

Note how obj.toString() generates 601BB1 for its *hashcode* value. A hashcode helps the JVM find data that might be anywhere in a large block of memory. In Java, hashcode values, like 601BB1, are written as hexadecimal numbers. We described the hexadecimal number system in Section 12.15. What follows is a review of that description.

Hexadecimal Numbers

Hexadecimal numbers use digits that can have one of 16 values—0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F (the lowercase letters a through f are also acceptable). The hexadecimal digits A through F represent the decimal numbers 10 through 15. With 16 unique digits, hexadecimal numbers form what is known as a base-16 number system. In the familiar base-10 number system for decimal numbers, suppose you're counting up and you get to the largest digit, 9. To form the next number, 10, you need two digits—a 1 at the left and a 0 at the right; the result is 10. Likewise, suppose that you're counting up with hexadecimal numbers and you get to the largest digit, F for 15. To form the next number, 16, you need two digits—a 1 at the left and a zero at the right; the result is 10. In other words, 10 is how you write 16 in hexadecimal. For additional help with hexadecimal counting, see Section 12.15 or Appendix 1. In Appendix 1, you'll see a sequence of hexadecimal numbers and their associated decimal numbers, in the context of the Unicode/ASCII character set. Appendix 8 explains how to make conversions back and forth between decimal and any other number system.

You know the hexadecimal number A is equivalent to the decimal number

10. What about the 601BB1 value generated by the previous code fragment—what is its equivalent decimal number? Appendix 8 explains the mathematics of converting any hexadecimal number to its decimal equivalent, but we'll present a shortcut. In Windows 10, for example, select Start > Calculator and navigate to Programmer. In the Programmer window, click the Hex button, enter 601BB1, and after DEC you'll see 6,298,545, which is the decimal equivalent of 601BB1. Thus, in the previous code fragment, when obj.toString() returns a string with 601BB1 at the right of the @ sign, it means the obj object's memory address is one of just a few memory addresses in the 6,298,545th row of an object *hash table.* The JVM can jump immediately to that row and, within that row, quickly perform a short search for the correct address.

Overriding the toString Method

Retrieving the class name, an @ sign, and a hashcode is usually worthless, so you'll almost always want to avoid calling the Object class's toString method and instead call an overriding toString method. The reason we're discussing the Object class's toString method is because it's easy to call it accidentally, and when that happens, we want you to understand what's going on.

Because the Object class defines a toString method, every class has a toString method, even if it does not define one or inherit one through some other class it explicitly extends. Many Java API classes define overriding toString methods. For example, the String class's toString method trivially returns the string that's stored in the String object. As described in Chapter 10, the ArrayList class's toString method (inherited from the

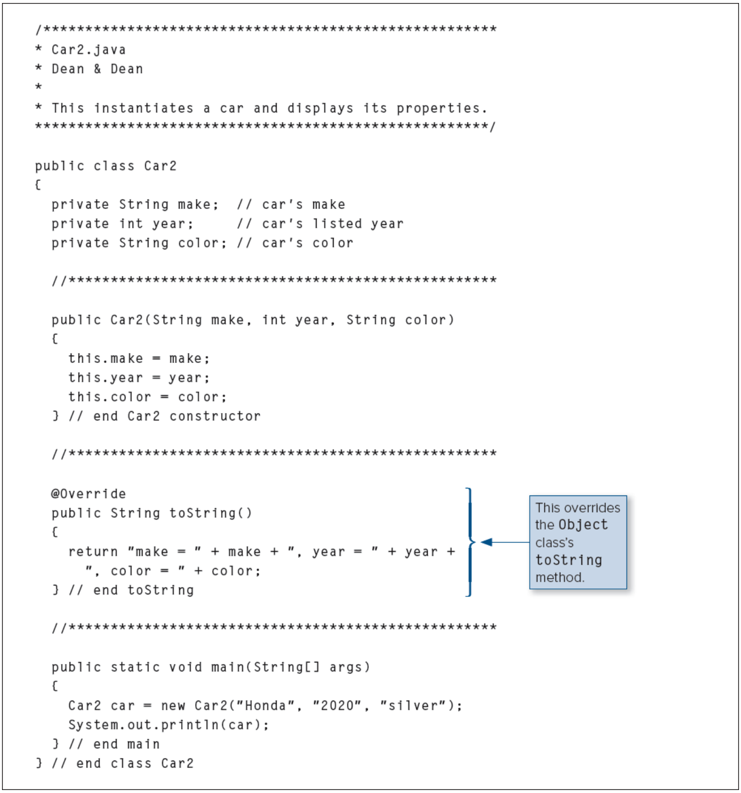
AbstractCollection class) returns a square-bracketed comma-delimited list of strings that represent the individual array elements. The Date class's toString method returns a Date object's month, day, year, hour, and second values as a single concatenated string. In general, toString methods should return a string that describes the calling object's contents.

Because retrieving the contents of an object is such a common need, you should get in the habit of providing an explicit toString method for most of your programmer-defined classes. Typically, your toString methods should simply concatenate the calling object's stored data and return the resulting ⚠ string. Your toString methods should not print the concatenated string value; they should just return it. We're mentioning this point because novice programmers have a tendency to put print statements in their toString methods, and that's wrong. A method should do only what it's supposed to do and nothing more. The toString method is supposed to return a string value, and that's it!

For example, look at the toString method in the Car2 program in Figure 14.2. It returns a string that describes the calling object's contents.

Implicit toString Method Calls

In the Car2 program, the main method has no explicit toString method call. So how does this program illustrate use of the toString method? Whenever a reference appears alone inside a print statement (System.out.print or system.out.println), the JVM automatically calls the referenced object's toString method. In Figure 14.2, this statement generates a call to the toString method in the Car2 class:



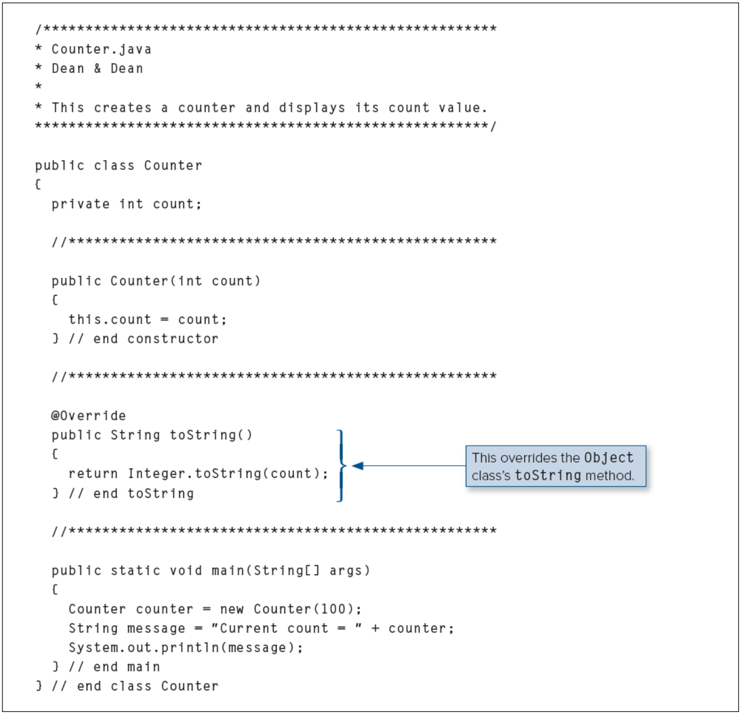
```
/*********************************************************
* Car2.java
* Dean & Dean
*
* This instantiates a car and displays its properties.
*********************************************************/

public class Car2
{
  private String make;  // car's make
  private int year;     // car's listed year
  private String color; // car's color

  //******************************************************

  public Car2(String make, int year, String color)
  {
    this.make = make;
    this.year = year;
    this.color = color;
  } // end Car2 constructor

  //******************************************************

  @Override
  public String toString()
  {
    return "make = " + make + ", year = " + year +
      ", color = " + color;
  } // end toString

  //******************************************************

  public static void main(String[] args)
  {
    Car2 car = new Car2("Honda", "2020", "silver");
    System.out.println(car);
  } // end main
} // end class Car2
```

This overrides the `Object` class's `toString` method.

Figure 14.2 Car2 program that illustrates overriding the toString method

System.out.println(car);

Let's look at another example that uses the toString method. See the Counter program in Figure 14.3. Once again, there's a toString method and no explicit call to it. So how does it get called? When you concatenate a reference variable and a string (with the + operator), the JVM automatically calls the reference's toString method. Thus, in Figure 14.3, this statement's counter reference generates a call to the Counter class's toString method:



```
/******************************************************
* Counter.java
* Dean & Dean
*
* This creates a counter and displays its count value.
******************************************************/

public class Counter
{
  private int count;

  //***************************************************

  public Counter(int count)
  {
    this.count = count;
  } // end constructor

  //***************************************************

  @Override
  public String toString()
  {
    return Integer.toString(count);
  } // end toString

  //***************************************************

  public static void main(String[] args)
  {
    Counter counter = new Counter(100);
    String message = "Current count = " + counter;
    System.out.println(message);
  } // end main
} // end class Counter
```

This overrides the `Object` class's `toString` method.

Figure 14.3 Counter program that illustrates implicitly calling the toString method

```
String message = "Current count = " + counter;
```

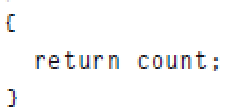
Note that you'll often see the toString method explicitly called with the standard call syntax even when it's not necessary. For example, in the Counter program's main method, we might have used this alternative implementation for the message assignment statement:

```
String message = "Current count = " + counter.toString();
```

Some programmers would claim that this alternative implementation is better because the code is more self-documenting. Some programmers would claim that the original implementation is better because the code is more compact. We don't have a preference as to which implementation is better—either way is fine.

Counter Program's toString Method—A Detailed Analysis

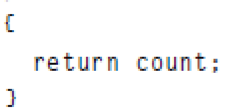
Let's revisit the toString method in Figure 14.3's Counter program. Because the Counter class contains only one piece of data, count, there's no need for concatenation code as part of the toString implementation. Just return count's value and that's it. So this might have been your first-cut implementation for toString:



```
public int toString()
{
  return count;
}
```

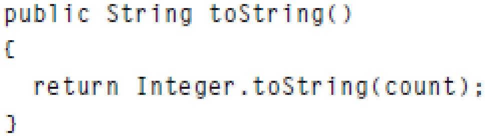
But this produces a compile-time error. Do you know why? An overriding method must have the same return type as the method it's overriding. Because the Counter class's toString method is an overriding ⚠ implementation of the Object class's toString method, the two methods must have the same return type.

Because the Object class's return type is a String, the above int return type generates an error. With that in mind, this might have been your second-cut implementation for toString:



```
public String toString()
{
  return count;
}
```

But this also produces an error. Why? Incompatible types. The returned value, count, is an int, and ⚠ the return type is defined to be a String. The solution is to convert count explicitly to a String before returning it, like this:

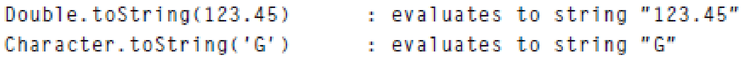


```
public String toString()
{
  return Integer.toString(count);
}
```

Do you understand the Integer.toString code? In Chapter 5, you learned that all primitive types have a corresponding wrapper class. Integer is one such class—it wraps up the int primitive. The Integer class's toString method returns a string representation of its passed-in int argument. So if count is 23, then Integer.toString(count) returns the string “23”.

Quick quiz: Is the Integer class’s toString method a static method or an instance method? Look at the method call’s prefix. The method call, Integer.toString, uses a class name for the prefix. When a method call uses a class name for a prefix instead of a reference variable, you know the method is a static method. Thus, Integer’s toString is a static method.

Note that all the wrapper classes have toString methods. They all do the same thing—they return a string representation of their passed-in argument. Here are some examples:

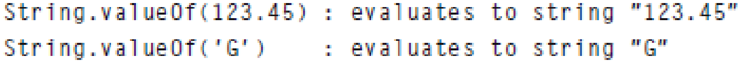


```
Double.toString(123.45)       : evaluates to string "123.45"
Character.toString('G')       : evaluates to string "G"
```

String’s valueOf Method

There’s another way to convert

primitive variables to strings. Use the String class's valueOf method. This takes a primitive value and returns a string. Like the wrapper toString methods described above, it's a static method, so you must use its class name, String, as a prefix. Thus, instead of the previous method calls, you could use these method calls:



```
String.valueOf(123.45) : evaluates to string "123.45"
String.valueOf('G')    : evaluates to string "G"
```

The valueOf method works with different data types because it's an overloaded method, and the JVM automatically selects that particular method whose parameter type matches the type of the data provided. Because we frequently customize toString methods, String's valueOf method is likely to produce more consistent standard conversions.

In addition to converting primitives to strings, the valueOf method can also be used to convert an array of vowel characters to a string. This code fragment prints the string “aeiou”:

```
char[] vowels = {'a', 'e', 'i', 'o', 'u'};
System.out.print(String.valueOf(vowels));
```

14.5 Polymorphism and Dynamic Binding

Polymorphism Overview

If you ask an object-oriented programming (OOP) aficionado to name the three most important characteristics of OOP, he or she will probably answer “encapsulation, inheritance, and polymorphism.” The previous chapter discussed encapsulation and inheritance. Now it’s time to discuss polymorphism. The word *polymorphism* means “having many forms.” It comes from the Greek root words *poly* (many) and *morph* (form). In chemistry and mineralogy, polymorphism is when a substance can crystallize in two or more alternative forms. In zoology, polymorphism is when a species has two or more different forms, like the different castes of bees spawned by the same queen to perform different functions in a beehive. In computer science, polymorphism is when different types of objects respond differently to the same method call.

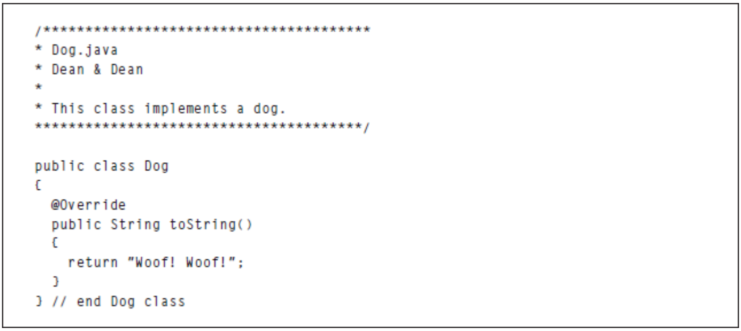
Here's how it works. You declare a general type of reference variable that is able to refer to objects of different types. What is the most general type of reference variable? It's an Object reference variable, declared, for example, like this:

```
Object obj;
```

Once you have declared a reference variable of type Object, you can use it to refer to any type of object. For example, suppose you define a class named Dog, as in Figure 14.4, and another class named Cat, as in Figure

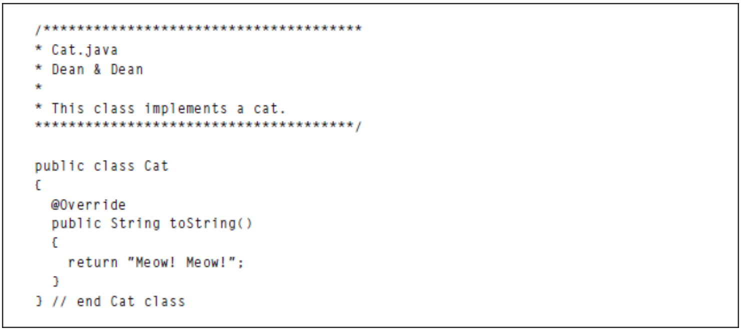
14.5. Each of the two derived classes contains a toString method that overrides the toString method in the Object class. Notice that the two toString methods shown override Object's toString method in different ways. One returns what a dog says, "Woof! Woof!," and the other returns what a cat says, "Meow! Meow!"

The different toString method definitions in the Dog and Cat classes enable the toString method to be polymorphic. If you call toString with a reference to a Dog object, it responds the way a dog would respond, but if you call toString with a reference to a Cat object, it responds the way a cat would respond. The driver in Figure 14.6 demonstrates this effect. Notice how the obj reference variable can contain a reference to either a Dog object or a Cat object, and that object determines which toString method is called.



```
/***************************************
* Dog.java
* Dean & Dean
*
* This class implements a dog.
***************************************/

public class Dog
{
  @Override
  public String toString()
  {
    return "Woof! Woof!";
  }
} // end Dog class
```

Figure 14.4 Dog class for the Pets program driven by code in Figure 14.6



```
/***************************************
* Cat.java
* Dean & Dean
*
* This class implements a cat.
***************************************/

public class Cat
{
  @Override
  public String toString()
  {
    return "Meow! Meow!";
  }
} // end Cat class
```

Figure 14.5 Cat class for the Pets program driven by code in Figure 14.6

Why does the program print “Woof! Woof!” twice? There are two print statements. The first one explicitly calls a toString method. The second one uses an implicit call to a toString method—when a reference variable appears alone in a String context, the compiler automatically appends

.toString() to the bare reference variable. So the last two statements in the Pets class are equivalent.

Dynamic Binding

The terms *polymorphism* and *dynamic binding* are intimately related, but they're not the same. Polymorphism is a form of behavior. Dynamic binding is the mechanism for that behavior—how it's implemented. Specifically, polymorphism is when different types of objects respond differently to the same method call. Dynamic binding is what the JVM does to match a polymorphic method call with a particular method.

Michael Pettigrew/Shutterstock

Just before the JVM executes a method call, it determines the type of the method call's actual calling object. If the actual calling object is from class X, the JVM *binds* class X's method to the method call. If the actual calling object is from class Y, the JVM binds class Y's method to the method call. After the JVM binds the appropriate method to the method call, the JVM executes the bound method. For example, note the obj.toString method call in the following statement near the bottom of Figure 14.6:

```
System.out.println(obj.toString());
```

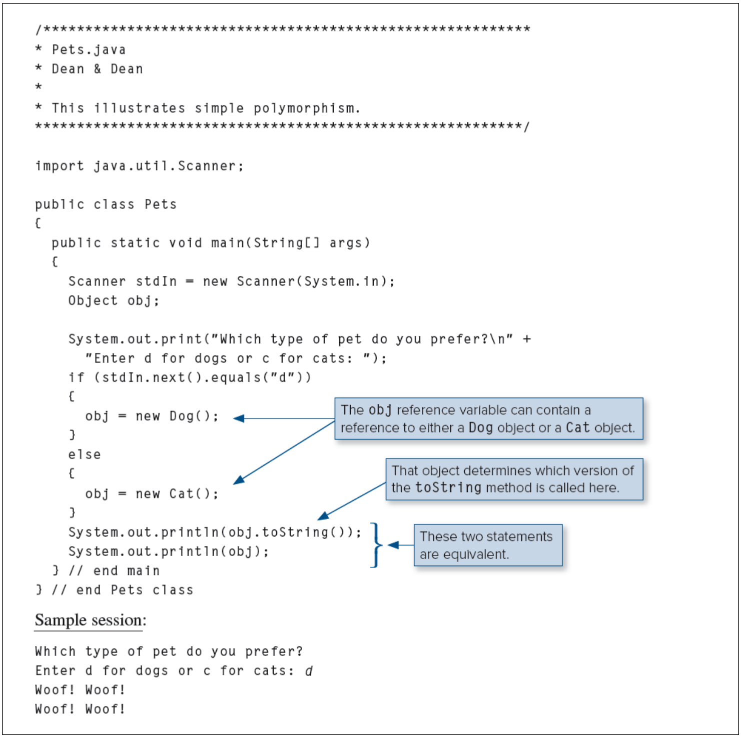
Depending on which type of object is referred to by obj, the JVM binds either Dog's toString method or Cat's toString method to the obj.toString method call. After binding takes place, the JVM executes the bound method and prints either "Woof! Woof!" or "Meow! Meow!"

Dynamic binding is "dynamic" because the JVM does the binding operation while the program is running. Binding takes place at the latest possible moment, right before the method is executed. That's why dynamic binding is often called *late binding*. Some programming languages bind method calls at compile time rather than at runtime. That is called *static binding*. Java's designers decided to go with dynamic binding rather than static binding because dynamic binding facilitates polymorphism.

Compilation Details

The Pets program illustrated polymorphic behavior by calling Dog and Cat versions of the toString method. Could we have done the same thing with Dog and Cat versions of a display method? In other words, if Dog implemented a display method that prints “I’m a dog,” would the following code work?

```
Object obj = new Dog();
obj.display();
```



```
/*************************************************************
* Pets.java
* Dean & Dean
*
* This illustrates simple polymorphism.
*************************************************************/

import java.util.Scanner;

public class Pets
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Object obj;

    System.out.print("Which type of pet do you prefer?\n" +
      "Enter d for dogs or c for cats: ");
    if (stdIn.next().equals("d"))
    {
      obj = new Dog();
    }
    else
    {
      obj = new Cat();
    }
    System.out.println(obj.toString());
    System.out.println(obj);
  } // end main
} // end Pets class
```

The `obj` reference variable can contain a reference to either a `Dog` object or a `Cat` object.

That object determines which version of the `toString` method is called here.

These two statements are equivalent.

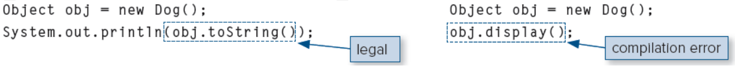
Sample session:

```
Which type of pet do you prefer?
Enter d for dogs or c for cats: d
Woof! Woof!
Woof! Woof!
```

Figure 14.6 Driver for the Pets program that includes classes in Figures 14.4 and 14.5

According to our dynamic binding discussion, the code would work just fine. The JVM would see a Dog object in the obj reference variable and bind the Dog's display method to the obj.display method call. But it doesn't matter that the code works fine in terms of dynamic binding. The code won't compile successfully because the compiler senses there might be a problem.

When the compiler sees a method call, *reference-variable.method-name*(), it checks to see if the reference variable's class contains a method definition for the called method. Note the obj.toString and obj.display method calls in the following examples. In the left example, the compiler checks to see if obj's class, Object, contains a toString method. Since it does, the code compiles successfully. In the right example, the compiler checks to see if obj's class, Object, contains a display method. Since the ⚠ Object class does not contain a display method, the code produces a compilation error.



```
Object obj = new Dog();
System.out.println(obj.toString());
```

```
Object obj = new Dog();
obj.display();
```

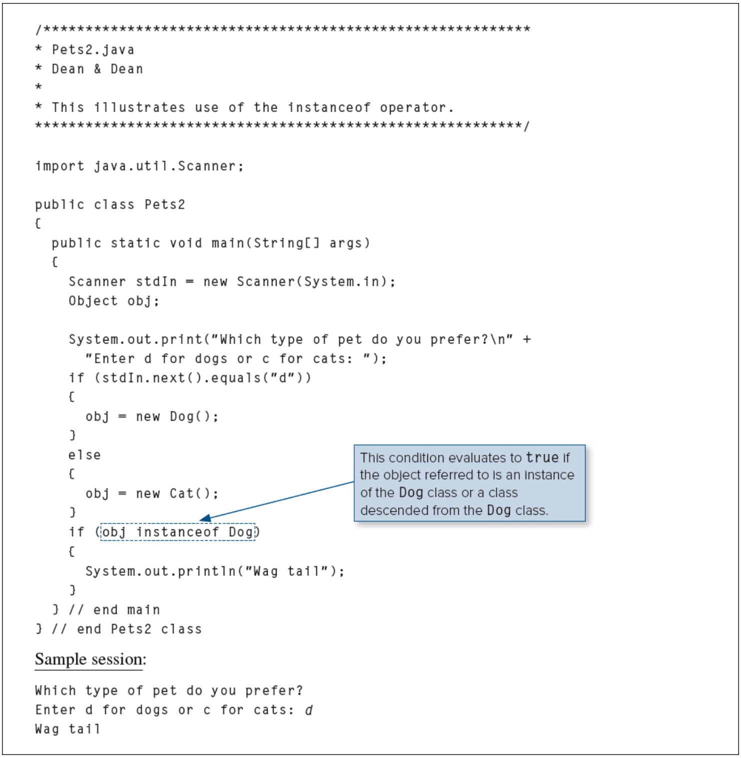
Does this mean that polymorphism works only for the methods defined in the Object class? Fortunately, that's not the case. Later in this chapter, you'll learn how to make polymorphism work for any method.

The instanceof Operator

Page 652

As you've seen, whenever a generic reference calls a polymorphic method, the JVM uses the type of the referenced object to decide which method to call. Figure 14.7 shows how you can do a similar thing explicitly in your code. In particular, suppose you want to see if a referenced object is an instance of some particular class. You can do this with a special operator called the instanceof operator (notice that the "o" in instanceof is lowercase). Using the Pets example again, suppose you want to print "Wags tail" if obj's object is an instance of class Dog or any class descended from class Dog. You can do that with the if statement at the bottom of the main method in

Figure 14.7. Thus, the instanceof operator provides a simple and direct way to sort out the various object types that might be referred to a by a generic reference variable.



```
/*************************************************************
* Pets2.java
* Dean & Dean
*
* This illustrates use of the instanceof operator.
*************************************************************/

import java.util.Scanner;

public class Pets2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Object obj;

    System.out.print("Which type of pet do you prefer?\n" +
      "Enter d for dogs or c for cats: ");
    if (stdIn.next().equals("d"))
    {
      obj = new Dog();
    }
    else
    {
      obj = new Cat();
    }
    if (obj instanceof Dog)
    {
      System.out.println("Wag tail");
    }
  } // end main
} // end Pets2 class
```

This condition evaluates to **true** if the object referred to is an instance of the **Dog** class or a class descended from the **Dog** class.

Sample session:

```
Which type of pet do you prefer?
Enter d for dogs or c for cats: d
Wag tail
```

Figure 14.7 Demonstration of the instanceof operator

14.6 Assignments When the Two Sides' Classes Are

Different

Let’s now look at something that’s quite common with polymorphic programs—assigning an object to a reference where the object’s class and the reference’s class are different. In the following code fragment, assume that Student is a subclass of Person. What does this code fragment do?

```
Person p = new Student();
Student s = new Person();
```

This generates a compile-time error.

The first line assigns a Student object (actually a reference to a Student object) to a Person reference variable. It’s assigning a subclass object to a superclass reference variable. That’s a legal assignment because a Student “is a” Person. It’s going up the inheritance hierarchy—the direction in which automatic ⚠ type promotion occurs. The second line tries to assign a Person object to a Student reference variable.

It's trying to assign a superclass object to a subclass reference variable. That's illegal because a Person is not necessarily a Student. The second line generates a compile-time error.

The "is a" mnemonic can help you remember the rule, but if you're a Curious George,2 you probably want more. If you want to understand the true rationale behind the rule, here goes. It's OK to assign a descendant-class object into an ancestor-class reference variable because all the compiler cares about is whether the assigned-in descendant-class object has all the members that any object of the reference variable's class should have. And if you assign a descendant-class object to an ancestor-class reference variable, it does. Why? Because descendant-class objects always inherit all ancestor-class members!

As with primitives, if there is compatibility, you can go the other way by using a cast. In other words, you can use a cast to force an object referred to by a more generic reference variable into a more specific type—a type that's below it in the same inheritance hierarchy. For example, if p is a Person reference variable, and Student inherits from Person, the compiler will accept this:

```
Student s = (Student) p;
```

Although the compiler will accept this statement, that does not necessarily mean the program will run successfully. For successful execution, when dynamic binding occurs, the object actually referred to by the p reference variable must be at least as specific as a Student. That is, the referenced object must be either an instance of the Student class or an instance of a descendant of the Student class. Why? Because after the assignment of the reference to a

Student reference variable, the object will be expected to have all of the members that a Student has, which is generally more than all the members a Person has.

14.7 Polymorphism with Arrays

So far, you've seen polymorphism in the context of code fragments and a simple Pets program. Those examples served their purpose—they illustrated the basics. But they didn't illustrate the real usefulness of polymorphism. The real usefulness of polymorphism comes when you have an array or ArrayList of generic reference variables and assign different types of objects to different elements. That allows you to step through the array or ArrayList and call a polymorphic method for each element. At runtime, the JVM uses dynamic binding to pick out the particular method that applies to each type of object found.

Polymorphism in an Explicit Inheritance Hierarchy

The Pets program used polymorphic toString methods for the Dog and Cat classes. The compiler accepted the Object reference variable with the toString method calls because the Object class defines its own toString method. Recall that polymorphism did not work for Dog and Cat display methods because the Object classs does not define its own display method. Suppose the method you want to make polymorphic is not defined in the Object classs. How can you have polymorphism and still satisfy the compiler? Actually, there are several related ways. One way is to create a superclass for the classes that define the different versions of the polymorphic method, and define the method within the superclass. Then use that superclass name when declaring the polymorphic reference variable(s). Another way to satisfy the compiler is to *declare the method* (specify the method heading only) in an abstract ancestor class and then use that ancestor class name for the reference variable type. Still another way to satisfy the compiler is to implement an *interface* that declares the method and then use that interface name for the reference variable type. We'll illustrate the first way in this section and the other two ways in subsequent sections.

Payroll Example

To illustrate polymorphism in an explicit inheritance hierarchy, we'll develop a payroll program that uses dynamic binding to select the appropriate method for calculating an employee's pay. Employees that happen to be salaried get dynamically bound to a Salaried class's getPay method. Employees that happen to be hourly get dynamically bound to an Hourly class's getPay method.

Let's start with the UML class diagram in Figure 14.8. It describes the Payroll program's class structure. Employee is a superclass and Salaried and Hourly are subclasses. The fourth class, Payroll, is the program driver. Its main method drives the Salaried and Hourly classes by instantiating them and then calling their methods. What is the association between Payroll and the other classes—inheritance or composition/aggregation? The UML class diagram's solid (black) diamonds indicate composition associations between the Payroll container and the Salaried and Hourly components. That should make sense when you realize that the Payroll class "has a" heterogeneous array of Salaried and Hourly objects. Assuming the Payroll class has exclusive control over these objects, its association with them is a composition, and the diamonds should be solid.

Before we get bogged down in the code for what is a fairly substantial program, let's look at how the program will work for some hypothetical employees. Simon is a salaried employee, with an annual salary of $48,000. The program pays salaried employees twice a month, so Simon gets paid $2,000 on the 15th and 30th of every month (the program makes the simplifying assumption that months have 30 days). Anna and Donovan are hourly employees, who work 8 hours a day, 5 days a week (Monday through Friday). With a pay rate of $25 per hour, Anna earns $1000 per week. With a pay rate of $20 per hour, Donovan earns $800 per week. If the first pay period spans Tuesday through Friday, with Friday occurring on the 4th, then this is what the program will print for a month's worth of paychecks:

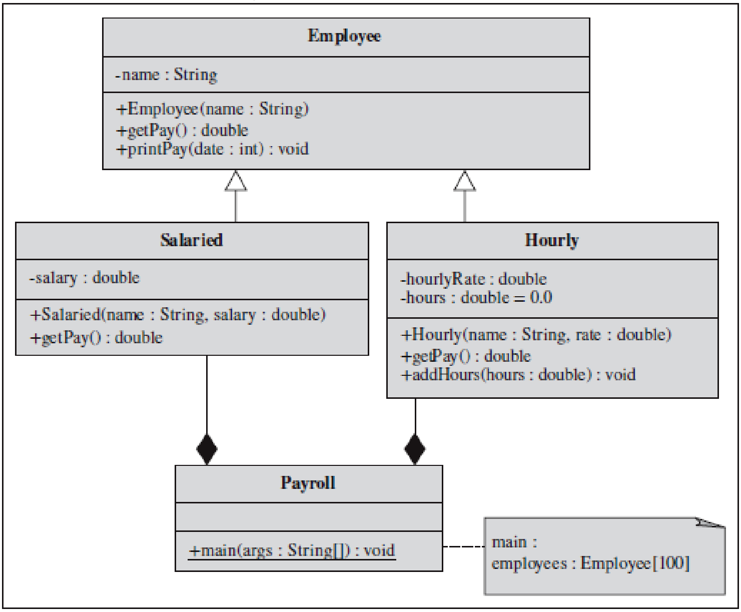
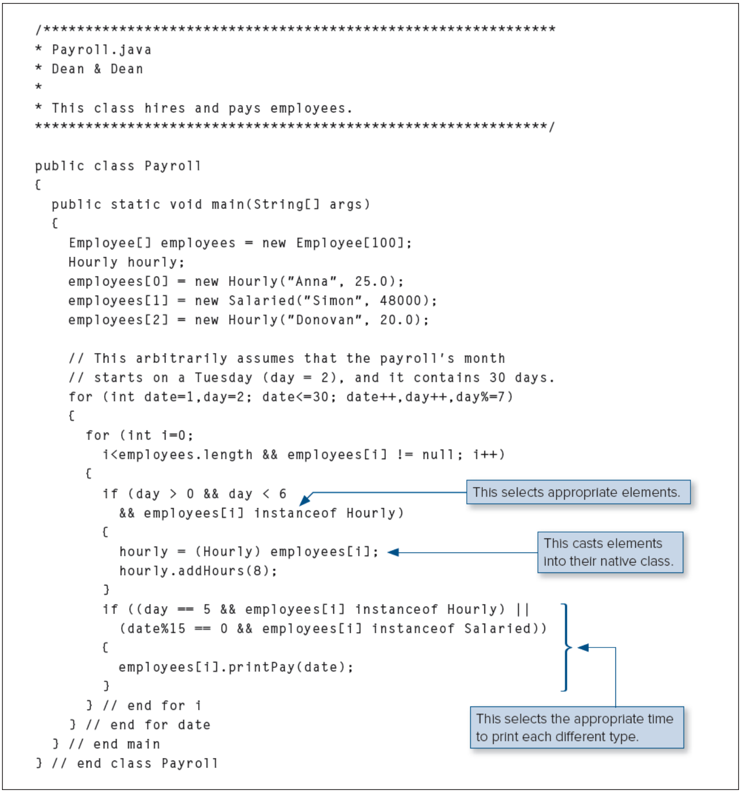

Employee
-name : String
+Employee(name : String)
+getPay() : double
+printPay(date : int) : void
Salaried
-salary : double
+Salaried(name : String, salary : double)
+getPay() : double
Hourly
-hourlyRate : double
-hours : double = 0.0
+Hourly(name : String, rate : double)
+getPay() : double
+addHours(hours : double) : void
Payroll
+main(args : String[]) : void
main :
employees : Employee[100]

Figure 14.8 Class diagram for the Payroll program

Output:

```
 4      Anna:    800.00
 4   Donovan:    640.00
11      Anna:   1000.00
11   Donovan:    800.00
15     Simon:   2000.00
18      Anna:   1000.00
18   Donovan:    800.00
25      Anna:   1000.00
25   Donovan:    800.00
30     Simon:   2000.00
```

Let's begin implementation with the main method in the driver in Figure 14.9. Note main's local variable, employees. It's declared to be a 100-element array of Employee objects. That's what it's declared as, but that's not exactly what it holds. As you can see from the assignment statements, the first three employees elements are an Hourly, a Salaried, and another Hourly. This is a heterogeneous array. All the elements in the array are instances of classes derived from the array's class, and none of them is an instance of the Employee class itself. Even though there may be no instances of the array's class in the array, the array's type is the right type to use because it is able to accommodate instances of all classes descended from the array's class.



```
/*************************************************************
* Payroll.java
* Dean & Dean
*
* This class hires and pays employees.
*************************************************************/

public class Payroll
{
  public static void main(String[] args)
  {
    Employee[] employees = new Employee[100];
    Hourly hourly;
    employees[0] = new Hourly("Anna", 25.0);
    employees[1] = new Salaried("Simon", 48000);
    employees[2] = new Hourly("Donovan", 20.0);

    // This arbitrarily assumes that the payroll's month
    // starts on a Tuesday (day = 2), and it contains 30 days.
    for (int date=1,day=2; date<=30; date++,day++,day%=7)
    {
      for (int i=0;
        i<employees.length && employees[i] != null; i++)
      {
        if (day > 0 && day < 6
          && employees[i] instanceof Hourly)
        {
          hourly = (Hourly) employees[i];
          hourly.addHours(8);
        }
        if ((day == 5 && employees[i] instanceof Hourly) ||
          (date%15 == 0 && employees[i] instanceof Salaried))
        {
          employees[i].printPay(date);
        }
      } // end for i
    } // end for date
  } // end main
} // end class Payroll
```

This selects appropriate elements.

This casts elements into their native class.

This selects the appropriate time to print each different type.

Figure 14.9 Driver for the simple Payroll program

Continuing with the main method, the outer for loop steps through 30 days, keeping track of two variables. Notice how the first compartment in the for loop header declares more than one variable of the specified type. The date variable represents the date of the month. It determines when salaried employees are paid. For simplicity, this program assumes 30 days per month. If you want to learn how to get the actual number of days in each month, go to Oracle's Java API website and read up on the Calendar class.[3] The day variable represents the day of the week. It determines when hourly employees are paid. Assuming that day 1 is a Monday, because the initial value of day is 2, the program's month starts on a Tuesday. Notice how the third compartment of the for loop header executes more than one operation. It increments both date and day and then uses day%=7 to make the day variable roll over to 0 whenever it reaches 7.

The inner for loop steps through the heterogeneous array of employees. The for loop header's second component employs a compound continuation condition. The i<employees.length condition alone would allow looping through all 100 elements of the employees array. What's the point of the for loop header's employees[i] != null condition? The program instantiates only three objects for this array, and 97 elements still contain the default value of null. If the program tries to call a method with a null reference, it crashes. More specifically, it generates a NullPointerException error the first time it tries to use the null reference. The employees[i] != null condition avoids that by stopping the loop when it gets to the first null element.

Inside the inner for loop, the first if statement accumulates hours for hourly workers. It checks to see if day is a weekday (not 0 or 6). It also checks to see if the object referenced by the current array element is an instance of the Hourly class. This enables the program to accumulate hours only during working days of the week and only for hourly workers. Once we know that the actual object is an instance of the Hourly class, it's safe to cast the generic reference into an Hourly reference. So we proceed to cast that reference into type Hourly, assign it to an Hourly reference variable, then use that specific type of reference variable to call the addHours method. Why did we jump through those hoops? Suppose we tried to call the addHours method with our generic reference in a statement like this:

```
employees[i].addHours(8);
```

The compiler would generate the following error message:

```
cannot find symbol

symbol : method addHours(int)
```

Because there is no addHours method in the Employee class, but there is one in the Hourly class, we must cast the array element explicitly into an Hourly type of reference and use that reference to call the method that we need.

Now look at the second if statement in the inner for loop. Instead of accumulating hours, its purpose is generating output for a payroll report. This if statement executes if either of two or'd conditions is true. The first condition is true if it's Friday (day = 5) and if the calling object is an instance of the Hourly class. The second condition is true if it's the middle of the month and if the calling object is not an instance of the Hourly class. If either of these conditions is satisfied, the raw array element calls the method like this:

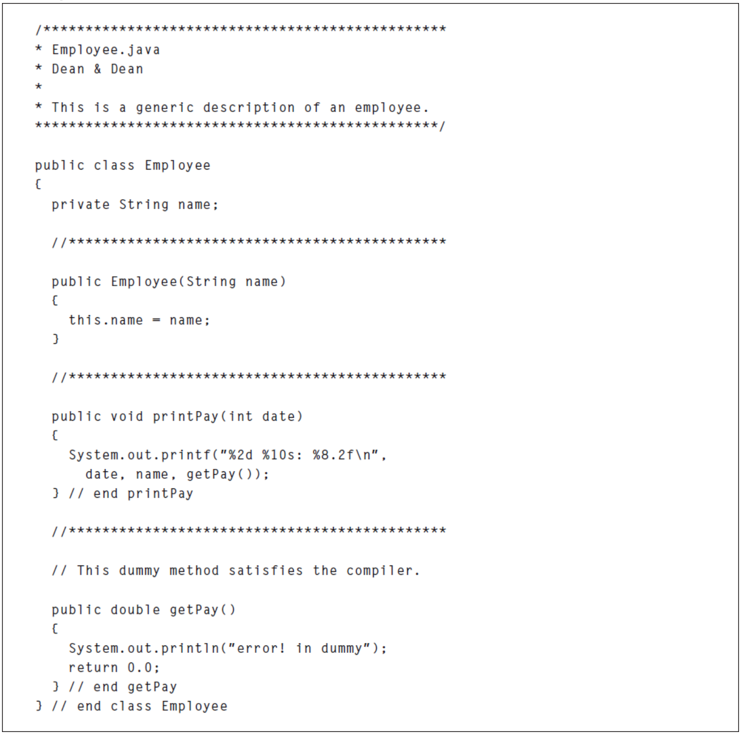
```
employees[i].printPay(date);
```

This strategy wouldn't have worked with the addHours method in the first if statement, but it does work with the printPay method in the second if statement. Why? Look at the UML specification of the Employee class in Figure 14.8. This time, the method being called, printPay, is supposed to be defined in the array's class.

Now let's work on implementation of that printPay method in the Employee class. In Figure 14.10, note how printPay prints the date and the employee's name and then calls getPay. The getPay method is supposed to calculate an employee's pay. But the Employee class's getPay method simply returns 0.0. What's up with that? Are employees really paid nothing? Certainly not! The Employee class's getPay method is simply a dummy method that's never executed. The "real" getPay methods (i.e., the ones that are executed) are the overriding definitions in the Salaried and Hourly subclasses. These overriding definitions make the getPay method polymorphic! How does the JVM know to use those methods and not the dummy method in the Employee class? When it performs dynamic binding, the JVM looks at the method's calling object. For the getPay case, the calling object is an instance of either the Salaried class or the Hourly class. Can you see why?

Go back to Figure 14.9's main method and note the assignment of an Hourly object into employees[0]. When employees[i].printPay() gets called with i equal to 0, the calling object is an Hourly object. Within the printPay method, when getPay is called, the calling object is still an Hourly object. Therefore, the JVM uses the Hourly class's getPay method. And that's what we want—the employees[0] object is an Hourly, so it uses the Hourly class's getPay method, which resets hours after each payment.

The same argument can be applied to the employees[1] object. Because it’s a Salaried object, it uses the Salaried class’s getPay method, which is independent of hours. Thanks to polymorphism and dynamic binding, life is good.



```
/****************************************************
* Employee.java
* Dean & Dean
*
* This is a generic description of an employee.
****************************************************/

public class Employee
{
  private String name;

  //**********************************************

  public Employee(String name)
  {
    this.name = name;
  }

  //**********************************************

  public void printPay(int date)
  {
    System.out.printf(”%2d %10s: %8.2f\n”,
      date, name, getPay());
  } // end printPay

  //**********************************************

  // This dummy method satisfies the compiler.

  public double getPay()
  {
    System.out.println(”error! in dummy”);
    return 0.0;
  } // end getPay
} // end class Employee
```

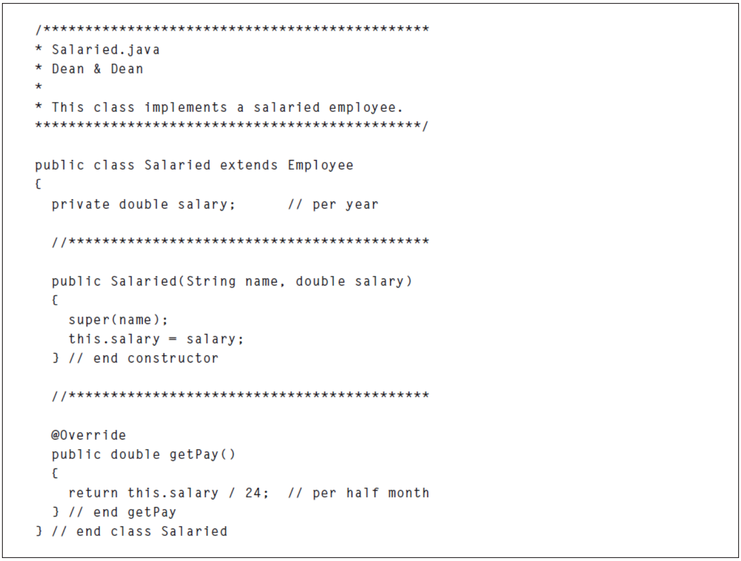
Figure 14.10 Employee class

The really cool thing about

polymorphism and dynamic binding is being able to program generically. In the main method, we can call printPay for all the objects in the array and not worry about whether the object is Hourly or Salaried. We just assume that printPay works appropriately for each employee. This ability to program generically enables programmers to think about the big picture without getting bogged down in details.

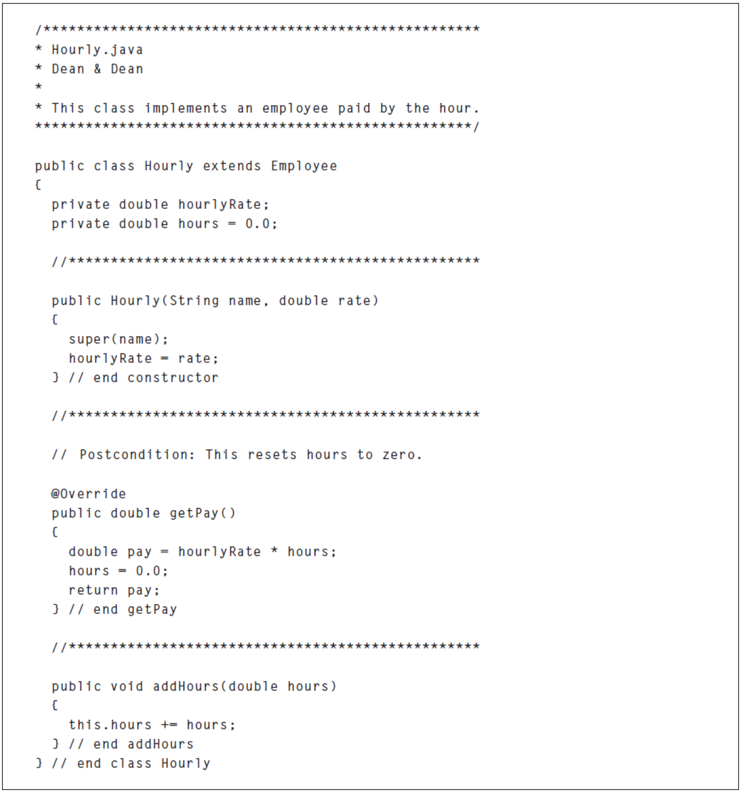
In the Employee class, were you bothered by the dummy getPay method? Were you thinking, “Why include a getPay method in the Employee class even though it’s never executed?” It’s needed because if there were no getPay method in the Employee class, the compiler would generate an error. Why? Because when the compiler sees a method call with no dot prefix, it checks to make sure that the method can be found within the current class. The getPay() method call (within the printPay method) has no explicit calling object, so the compiler requires the Employee class to have a getPay method.

Now it's time to implement the "real" getPay methods. See Figures 14.11 and 14.12. The methods in these two classes are both simple, but they are different. To keep the JVM from selecting the dummy getPay method in the base class during dynamic binding, all derived classes should override that method.



```
/*************************************************
* Salaried.java
* Dean & Dean
*
* This class implements a salaried employee.
*************************************************/

public class Salaried extends Employee
{
  private double salary;      // per year

  //**********************************************

  public Salaried(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //**********************************************

  @Override
  public double getPay()
  {
    return this.salary / 24;  // per half month
  } // end getPay
} // end class Salaried
```

Figure 14.11 Salaried class

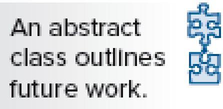


```
/*****************************************************
* Hourly.java
* Dean & Dean
*
* This class implements an employee paid by the hour.
*****************************************************/

public class Hourly extends Employee
{
  private double hourlyRate;
  private double hours = 0.0;

  //**************************************************

  public Hourly(String name, double rate)
  {
    super(name);
    hourlyRate = rate;
  } // end constructor

  //**************************************************

  // Postcondition: This resets hours to zero.

  @Override
  public double getPay()
  {
    double pay = hourlyRate * hours;
    hours = 0.0;
    return pay;
  } // end getPay

  //**************************************************

  public void addHours(double hours)
  {
    this.hours += hours;
  } // end addHours
} // end class Hourly
```

Figure 14.12 The Hourly class

14.8 abstract Methods and Classes

The dummy getPay method in Figure 14.10 is an example of a *kludge* (pronounced “klooj”). Kludgy code is ugly, inelegant code that provides a workaround for a problem. Usually, inelegant code is hard to understand. And hard-to-understand code is hard to maintain. So try to avoid kludges. Sometimes that's not possible, but in this case, we can indeed avoid the dummy-method kludge. Here's how. . . .

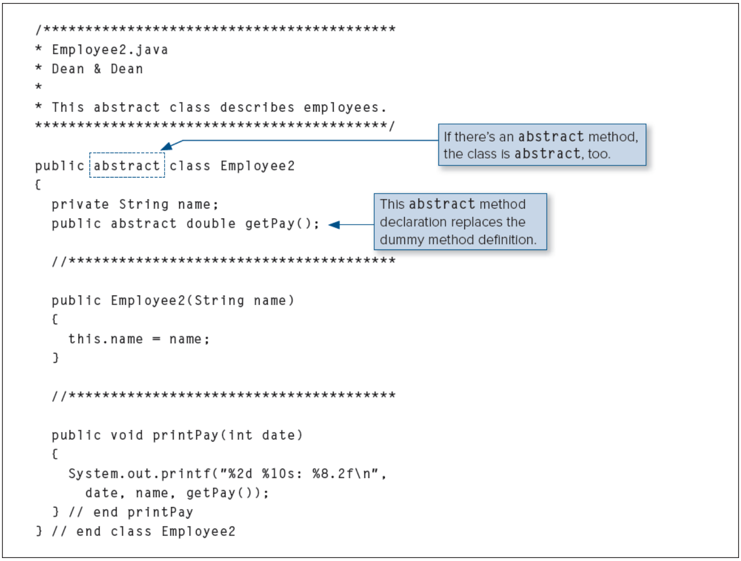


An abstract class outlines future work.

If you find yourself writing a dummy method that will be overridden by methods defined in all instantiable descendant classes, stop and reconsider. There's a better way. Use an abstract class to tell the compiler what you're trying to do ahead of time. In the abstract class, provide an abstract method declaration for each method that is merely a "dummy" method for overriding method(s) in subclass(es). To declare a method, just write the method heading with the additional modifier abstract, and terminate this modified method heading with a semicolon. For example, note the abstract modifier for the getPay method in the Employee2 class in Figure 14.13.

An abstract declaration doesn't contain enough information to define the method. It just specifies its outside-world interface and says that definition(s) will exist somewhere else. Where? In all instantiable descendant classes! Using an abstract method avoids the inelegant dummy method definition, and it's a better way to implement polymorphism.

The abstract modifier is well named. Something is abstract if it is general in nature, not detailed. An abstract method declaration is general in nature. It doesn't provide method details. It just serves notice that the method exists and that it must be fleshed out by “real” method definitions in all instantiable descendant classes. Have we followed this rule for our program? In other words, do we have definitions of the getPay method in all the Employee2 descendant classes? Yes, the Salaried and Hourly classes in Figures 14.11 and 14.12 already contain the required getPay method definitions. However, we need to revise the Salaried, Hourly, and Payroll classes by making these replacements:



```
/*********************************************
* Employee2.java
* Dean & Dean
*
* This abstract class describes employees.
*********************************************/

public abstract class Employee2
{
  private String name;
  public abstract double getPay();

  //*****************************************

  public Employee2(String name)
  {
    this.name = name;
  }

  //*****************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n",
      date, name, getPay());
  } // end printPay
} // end class Employee2
```

If there’s an `abstract` method, the class is `abstract`, too.

This `abstract` method declaration replaces the dummy method definition.

Figure 14.13 Employee2 class, using the

abstract modifier to replace a dummy method definition with a simpler method declaration

```
Employee → Employee2
Salaried → Salaried2
Hourly → Hourly2
Payroll → Payroll2
```

Then the Salaried2, Hourly2, and Payroll2 classes will start out looking like this:

```
public class Salaried2 extends Employee2
{
  ...

public class Hourly2 extends Employee2
{
  ...

public class Payroll2
{
  public static void main(String[] args)
  {
    Employee2[] employees = new Employee2[100];
    ...
```

Here's another thing to note when declaring an abstract method. Because an abstract method declaration does not provide a definition for that method, the class definition is incomplete. Because the class definition is incomplete, it can’t be used to construct objects. The compiler recognizes this and complains if you don’t recognize it in your code. To satisfy the compiler, you must add an abstract modifier to the class heading whenever you have a class that contains one or more abstract methods. For example, note the abstract modifier in the Employee2 class heading in Figure 14.13.

Adding an abstract modifier to a class heading makes it impossible to instantiate an object from that class. If a program attempts to instantiate an abstract class, the compiler generates an error. For example, because Employee2 is an abstract class, we’d get a compilation error if we had a main method like this:

```
public static void main(String[] args)
{
  Employee2 emp = new Employee2("Benji");
}
```

Because `Employee2` is `abstract`, this generates a compilation error.

Sometimes you don’t want a subclass

to define a method that was declared to be abstract in its superclass. Instead you want to defer the method definition to the next generation. It's easy to do this. In the subclass, just ignore that method and declare the subclass abstract also (because at least that method is still undefined). You can defer method definitions like this as far as you want, provided you ultimately define them all in any nonabstract descendant class you use to instantiate objects.

We have said that if any method in a class is abstract, that class must be abstract. But this does not mean all methods in an abstract class must be abstract. It's frequently useful to include one or more non-abstract method definitions in an abstract class. Thus, classes descended from an abstract class can inherit non-abstract methods from that class and are not required to redefine those non-abstract methods. Do you recall the admonition in the margin in the previous chapter's Section 13.3? It said, "Plan to instantiate leaves only." You can enforce this plan by declaring all intended nonleaf classes to be abstract. Although they cannot be instantiated, these abstract classes can reduce programming work and improve consistency by defining common, reusable nonabstract methods.

Illegal to Use private, final, or static with abstract

An abstract method declaration cannot be private, and the definitions of the method that appear in ⚠ descendant classes cannot be private either. Why? An abstract method declaration must mirror the headings for all its associated overriding methods. So if an abstract method declaration were allowed to be private, then its associated overriding methods would have to be private too. But private doesn't make sense for methods that override abstract methods. After all, the point of overriding an abstract method is to enable a superclass to call the subclass's overriding method. For example, in Figure 14.13, the overriding method getPay is called from printPay, which is in the Employee2 superclass. That sort of call would be impossible if the subclass method were private. Because private doesn't make sense for methods that override abstract methods, Java's designers decided that private is not allowed for abstract method declarations and for methods that override abstract methods.

An abstract class or method cannot be final. The final modifier keeps a class from being extended and keeps a method from being overridden. But an abstract class is supposed to be extended and an abstract method is supposed to be overridden, so it's illegal to use final with abstract.

It's illegal to use the static modifier with an abstract method. The purpose of having an abstract method is to support polymorphic method calls, and polymorphic method calls work only for instance methods, not static methods. Thus, it makes no sense to allow abstract methods to be static methods.

14.9 Interfaces

An abstract class serves as a repository for variables and methods that are common to several subclasses. In this section, we describe *interfaces*, which are similar to abstract classes. In Chapter 10, you learned about the ArrayList and LinkedList classes, and you were briefly introduced to the List interface, which the two classes rely on. An interface is a template for designing classes that share certain qualities. The ArrayList and LinkedList classes both implement lists, and, as such, they share certain qualities common to all lists. Specifically, they share many of the same method declarations. To ensure consistency between the methods, the classes are built using the List interface as a template.

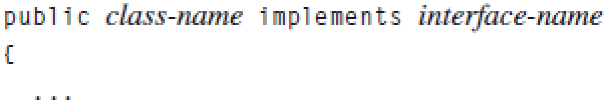
Using an Interface as a Contract to Ensure Consistency

There are several different ways to use an interface. Let's start with its most popular purpose, which was alluded to in the prior paragraph. Think of an interface as a contract between a program designer and program implementers that standardizes communication among different classes. This use of interfaces is essential to the success of large programming projects. Suppose, for example, that you are designing an accounting system, and you’re currently focusing on “asset” accounts, which keep track of the value of things the company owns or has rights to. Typical asset accounts are Cash, Accounts Receivable, Inventory, Furniture, Manufacturing Equipment, Vehicles, Buildings, and Land. These things are different from each other, so it would not be natural for classes representing them to be in a single inheritance hierarchy. Some of these accounts (Furniture, Manufacturing Equipment, Vehicles, and Buildings) describe long-term or “fixed” assets whose values depreciate gradually over time. Each year, an accountant prepares a set of financial statements, like the Balance Sheet and a Profit and Loss Statement. This preparation requires access to information like original cost, date of acquisition, and depreciation rate for objects that represent depreciating assets.

Page 664

To facilitate such access, references to these objects could be in an array or ArrayList. Then a program could step through that array or ArrayList and call identically named polymorphic get methods to retrieve originalCost, acquisitionDate, and depreciationRate values from objects that represent depreciating assets. Suppose different programmers are writing the classes for the different accounts. The best way to ensure that the different classes use the get methods consistently is to have the classes use the same Java interface. In our accounting system example, the interface for the get methods that access originalCost, acquisitionDate, and depreciationRate instance variables might be called the AssetAging interface. The AssetAging interface would contain declarations for its methods, but not definitions.

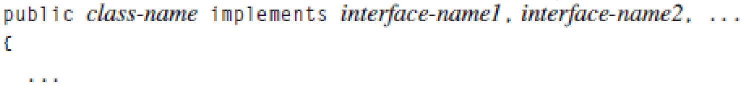
If a particular class includes method definitions for all of the methods declared in a particular interface (like AssetAging), you can tell the world (and the Java compiler) that that class provides such definitions by appending an implements clause to its class heading, like this:



```
public class-name implements interface-name
{
  ...
```

When a class has an implements clause, you can say that the class “implements the interface.” Sometimes, implementing an interface is referred to as *interface inheritance*, or simply inheritance.

If you want a class to implement multiple interfaces, you include a comma-separated list of the interface names in the class heading, like this:



```
public class-name implements interface-name1, interface-name2, ...
{
  ...
```

If you want a class to inherit from a superclass and also to implement an interface, you include an extends clause and an implements clause, like this:

```
public class-name extends superclass-name implements interface-name
{
  ...
```

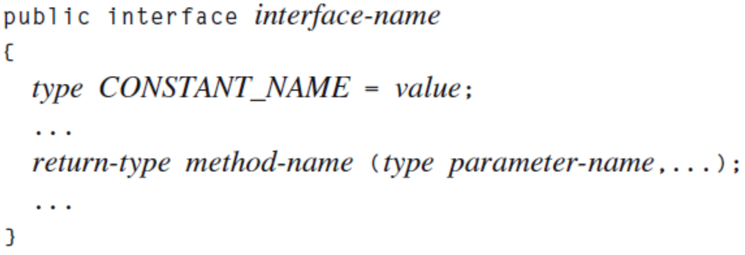
Multiple Inheritance

In describing an interface as implementing a contract that another class has to abide by, that should have sounded familiar. An abstract class also serves as a contract for other classes. For example, in the prior section the Employee2 abstract class served as a superclass contract for the Salaried2 and Hourly2 subclasses. So why not just always use an abstract class rather than an interface? There’s an important difference that makes an interface a better contract than an abstract class. In some programs, you’ll have a class that needs to follow the constraints of more than one template. In Java, a class can inherit from only one superclass, so if more than one template is needed, you can't use classes, not even if they're abstract. But you can use multiple interfaces for the multiple template contracts. That sort of thing is sometimes referred to as *multiple inheritance*. In other programming languages, multiple inheritance can be implemented with multiple superclasses for one subclass, but that can lead to confusion, and that's why Java's designers decided not to allow it.

What's the rationale for allowing multiple inheritance with interfaces, but not with superclasses? If it were allowed with superclasses, you could define a same-signature method in each of the two superclasses. That caveat applies to abstract classes as well as regular classes because an abstract class can define regular methods along with its abstract methods. If an object of the subclass attempts to call the method that's defined in both superclasses, it would be difficult for the JVM to figure out which method to bind to and execute. But with an interface, there are no regular methods, so if an object's two interfaces declare a same-signature method, and the object calls that method, the JVM knows to bind to the object's method, and there's no confusion about trying to bind to the interfaces' methods.

Interface Syntax

Here's the syntax for the most common form of an interface:



```
public interface interface-name
{
  type CONSTANT_NAME = value;
  ...
  return-type method-name (type parameter-name,...);
  ...
}
```

And here's syntax that is functionally equivalent:

```
public abstract interface interface-name
{
  public static final type CONSTANT_NAME = value;
  ...
  public abstract return-type method-name (type parameter-name,...);
  ...
}
```

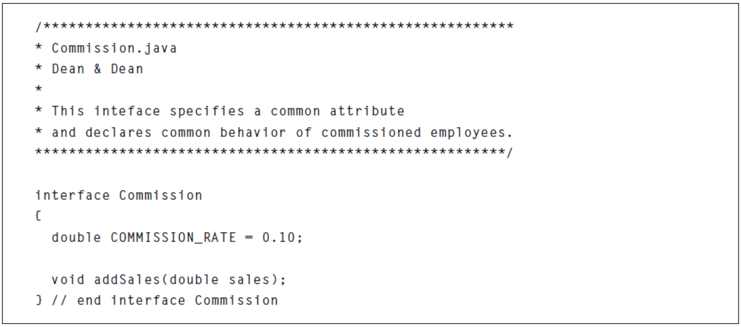
As you can see in the above syntax examples, you are allowed to include the abstract modifier in your interface heading, but it’s unnecessary. If you omit it, it’s inserted automatically behind the scenes. It’s omitted on Oracle’s documentation web pages, and that’s why we omit it in our upcoming examples.

The only type of field allowed in an interface is a public static named constant. No instance variables, no static variables, no private fields. You are allowed to include the public static final modifiers at the left of your interface named constant declarations, but it’s unnecessary. If you omit them, they’re inserted automatically behind the scenes. In following Oracle’s lead, we omit them in our upcoming examples.

There are three types of methods allowed in an interface—abstract methods, static methods, and default methods. You've already seen abstract methods and static methods. We'll describe default methods later in this section. All three method types are public by default. You are allowed to include the public modifier at the left of your interface method headings, but it's unnecessary. If you omit it, it's inserted automatically behind the scenes. In following Oracle's lead, we omit it in our upcoming examples. As you can see in the above syntax examples, if you omit the abstract modifier, the method is an abstract method by default.

Payroll Program with Multiple Inheritance

Let's put what you've learned about interfaces into practice in the context of a complete program. We'll add multiple inheritance to the previous Payroll program with the help of an interface and two classes of commissioned employees.[4] One of those classes gets a "straight" commission. The other class gets a salary plus a commission. In both cases, the commission is based on a common fixed percentage of sales. Figure 14.14 contains the code for an interface that defines this fixed percentage as a named constant and declares a method that must be defined in all classes that implement the interface.



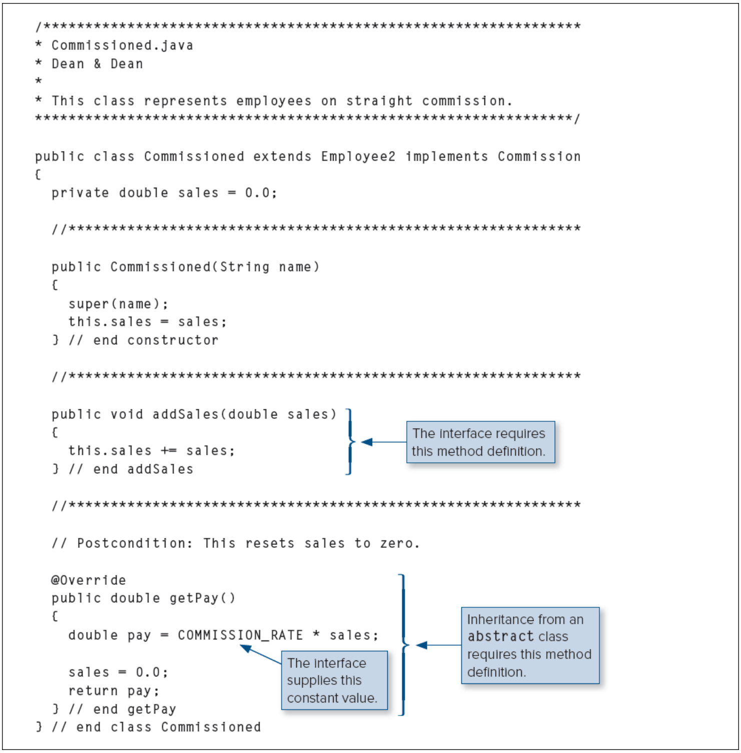
```
/********************************************************
* Commission.java
* Dean & Dean
*
* This inteface specifies a common attribute
* and declares common behavior of commissioned employees.
********************************************************/

interface Commission
{
  double COMMISSION_RATE = 0.10;

  void addSales(double sales);
} // end interface Commission
```

Figure 14.14 Commission interface, for an enhanced version of the Payroll

program

Figure 14.15 shows the code for a Commissioned class, which describes a class of employees who work on a straight commission. The Commissioned class extends Figure 14.13's Employee2 class. Employee2 is an abstract class, and as such, the Commissioned subclass must define all Employee2's abstract methods. The only abstract method in the Employee2 class is the getPay class, so the Comissioned class must define the getPay method, and yes, it does. This increases the total number of polymorphic getPay methods to three. In the commissioned class's heading, note the clause, implements Commission. This provides direct access to the COMMISSION_RATE named constant, which the Commissioned class's getPay method uses to do its job. When it implements the Commission interface, the

Commissioned class also takes on an obligation. It must define all the methods declared in that interface. The only method declared in the Commission interface is the addSales method, and yes, the Commissioned class defines this method, too.



```
/*******************************************************************
* Commissioned.java
* Dean & Dean
*
* This class represents employees on straight commission.
*******************************************************************/

public class Commissioned extends Employee2 implements Commission
{
  private double sales = 0.0;

  //*****************************************************************

  public Commissioned(String name)
  {
    super(name);
    this.sales = sales;
  } // end constructor

  //*****************************************************************

  public void addSales(double sales)
  {
    this.sales += sales;
  } // end addSales

  //*****************************************************************

  // Postcondition: This resets sales to zero.

  @Override
  public double getPay()
  {
    double pay = COMMISSION_RATE * sales;

    sales = 0.0;
    return pay;
  } // end getPay
} // end class Commissioned
```

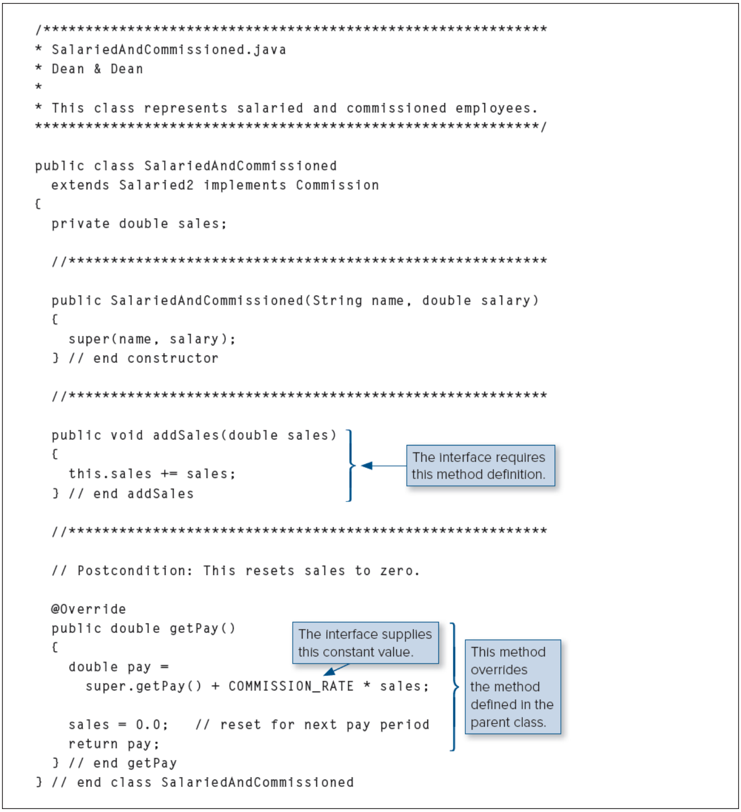
The interface requires this method definition.

Inheritance from an `abstract` class requires this method definition.

The interface supplies this constant value.

Figure 14.15 Commissioned class, for straight-commission employees in enhanced Payroll program

Figure 14.16 shows the code for a SalariedAndCommissioned class. This class extends the Salaried2 class. The Salaried2 class is like the Salaried class in Figure 14.11, except for one difference: whereas the Salaried class extends Employee, the Salaried2 class extends Employee2. The SalariedAndCommissioned class describes a class of employees that earn a salary and a commission. The Salaried2's class defines a getPay method, so the compiler does not insist that the SalariedAndCommissioned class also define a getPay method, but logically, we need to override the Salaried2 getPay method. Notice how the overriding method uses the super prefix to call the method that it overrides. This additional getPay method definition increases the total number of polymorphic getPay methods to four.



```
/*************************************************************
* SalariedAndCommissioned.java
* Dean & Dean
*
* This class represents salaried and commissioned employees.
*************************************************************/

public class SalariedAndCommissioned
  extends Salaried2 implements Commission
{
  private double sales;

  //**********************************************************

  public SalariedAndCommissioned(String name, double salary)
  {
    super(name, salary);
  } // end constructor

  //**********************************************************

  public void addSales(double sales)
  {
    this.sales += sales;
  } // end addSales

  //**********************************************************

  // Postcondition: This resets sales to zero.

  @Override
  public double getPay()
  {
    double pay =
      super.getPay() + COMMISSION_RATE * sales;

    sales = 0.0;    // reset for next pay period
    return pay;
  } // end getPay
} // end class SalariedAndCommissioned
```

The interface requires this method definition.

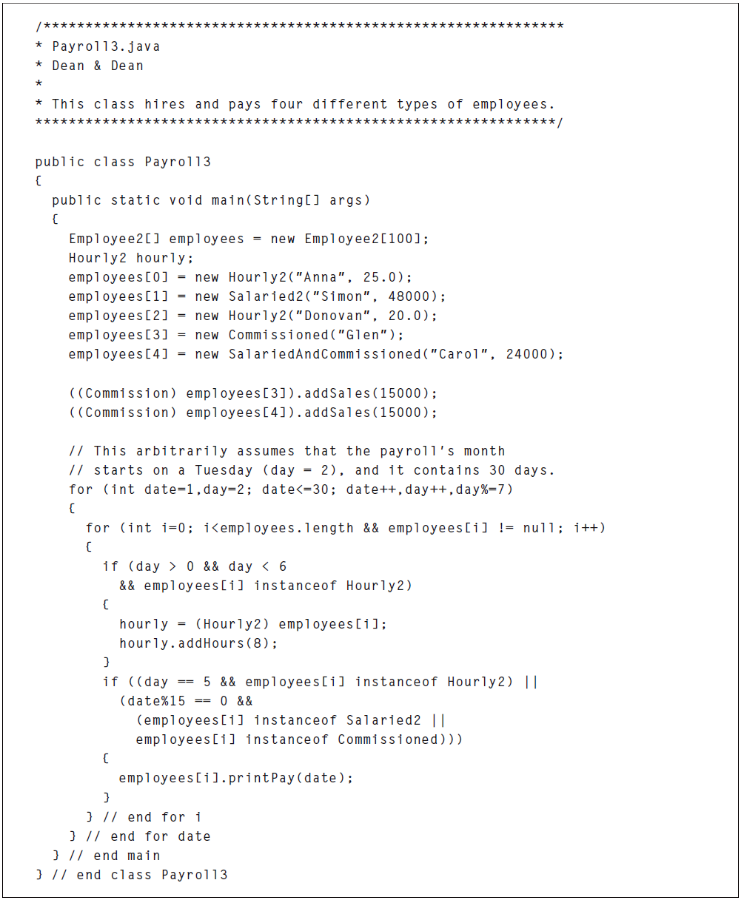
The interface supplies this constant value.

This method overrides the method defined in the parent class.

Figure 14.16 SalariedAndCommissioned class, for employees in enhanced Payroll program

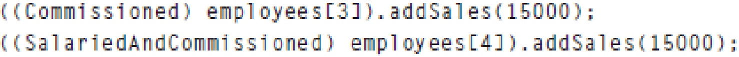
The SalariedAndCommissioned class also implements the Commission interface. This provides direct access to the COMMISSION_RATE named constant, which the getPay method uses to do its job. Because it implements the Commission interface, the SalariedAndCommissioned class must define all methods declared in that interface, and yes, it does define the addSales method.

To execute those additional classes, we need a Payroll3 class like that shown in Figure 14.17. The Payroll3 class adds two more objects (Glen and Carol) to the array. Then it uses those objects to call the addSales methods in the new classes. To make these method calls, we cast the array elements into the interface type. The compiler requires a cast because the addSales method does not appear in the Employee2 class. Note that we need an extra set of parentheses surrounding the (Commission) cast operator and the calling object. We could have used more specific casts like this:



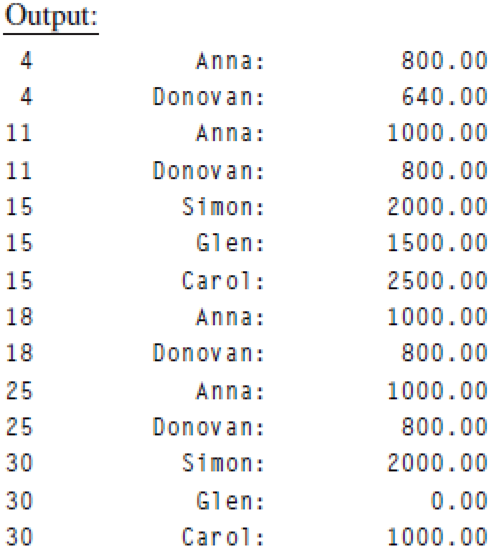
```
/*************************************************************
* Payroll3.java
* Dean & Dean
*
* This class hires and pays four different types of employees.
*************************************************************/

public class Payroll3
{
  public static void main(String[] args)
  {
    Employee2[] employees = new Employee2[100];
    Hourly2 hourly;
    employees[0] = new Hourly2("Anna", 25.0);
    employees[1] = new Salaried2("Simon", 48000);
    employees[2] = new Hourly2("Donovan", 20.0);
    employees[3] = new Commissioned("Glen");
    employees[4] = new SalariedAndCommissioned("Carol", 24000);

    ((Commission) employees[3]).addSales(15000);
    ((Commission) employees[4]).addSales(15000);

    // This arbitrarily assumes that the payroll's month
    // starts on a Tuesday (day = 2), and it contains 30 days.
    for (int date=1,day=2; date<=30; date++,day++,day%=7)
    {
      for (int i=0; i<employees.length && employees[i] != null; i++)
      {
        if (day > 0 && day < 6
          && employees[i] instanceof Hourly2)
        {
          hourly = (Hourly2) employees[i];
          hourly.addHours(8);
        }
        if ((day == 5 && employees[i] instanceof Hourly2) ||
          (date%15 == 0 &&
            (employees[i] instanceof Salaried2 ||
            employees[i] instanceof Commissioned)))
        {
          employees[i].printPay(date);
        }
      } // end for i
    } // end for date
  } // end main
} // end class Payroll3
```

Figure 14.17 Driver for the third version of the Payroll program



```
((Commissioned) employees[3]).addSales(15000);
((SalariedAndCommissioned) employees[4]).addSales(15000);
```

But it's more elegant to cast into the more generic Commission interface type and let the JVM select among the polymorphic alternatives as it does its dynamic binding. Using either type of casting, here's what the Payroll3 driver generates:



Output:

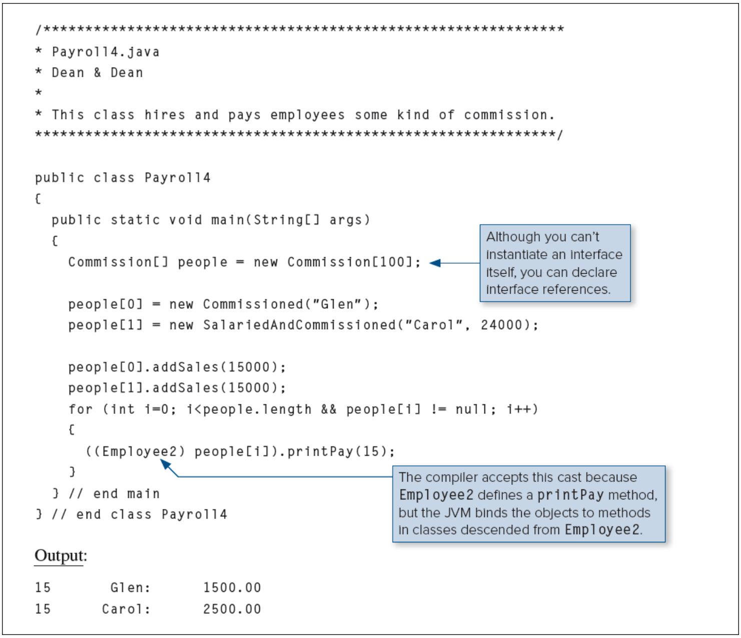
```
 4         Anna:       800.00
 4      Donovan:       640.00
11         Anna:      1000.00
11      Donovan:       800.00
15        Simon:      2000.00
15         Glen:      1500.00
15        Carol:      2500.00
18         Anna:      1000.00
18      Donovan:       800.00
25         Anna:      1000.00
25      Donovan:       800.00
30        Simon:      2000.00
30         Glen:         0.00
30        Carol:      1000.00
```

In our coded examples, notice the similarity between the use of an interface name and the use of a class name. It's not possible to instantiate an interface because it's inherently abstract, but you can use it like any ordinary class to specify type. For example, you can declare an array of elements whose type is an interface name, you can populate that array with instances of classes that implement that interface, and then you can pull objects out of that array and cast them into any type (class or interface) that those objects conform to. The Payroll4 driver in Figure 14.18 and the subsequent output illustrate these possibilities.

The trick is to think about what the compiler needs and what the JVM does. For example, you can create an array of interface references because the elements in the array are just references, not instantiated objects. The compiler lets you populate that array with references to objects from classes that implement that interface because it knows those objects can call any method the interface declares. In a method call, the compiler lets you cast a reference into the type of any class that declares or defines any version of that method because it knows the JVM can find at least one method to bind. At runtime, the JVM selects the most appropriate method to bind.

Static Methods and Default Methods

Prior to Java 8, static methods and default methods were not allowed in interfaces. Oracle added them because they served a need, and they preserved the interface construct's ability to implement multiple inheritance effectively. By "effectively," we mean they don't introduce confusion in terms of the JVM figuring out which inherited method to bind to and execute.



```
/*******************************************************************
* Payroll4.java
* Dean & Dean
*
* This class hires and pays employees some kind of commission.
*******************************************************************/

public class Payroll4
{
  public static void main(String[] args)
  {
    Commission[] people = new Commission[100];

    people[0] = new Commissioned("Glen");
    people[1] = new SalariedAndCommissioned("Carol", 24000);

    people[0].addSales(15000);
    people[1].addSales(15000);
    for (int i=0; i<people.length && people[i] != null; i++)
    {
      ((Employee2) people[i]).printPay(15);
    }
  } // end main
} // end class Payroll4
```

Although you can't instantiate an interface itself, you can declare interface references.

The compiler accepts this cast because `Employee2` defines a `printPay` method, but the JVM binds the objects to methods in classes descended from `Employee2`.

Output:

```
15       Glen:     1500.00
15      Carol:     2500.00
```

Figure 14.18 Demonstration of class-like properties of an interface

Page 671

To implement a static method in an interface, simply add the static modifier at the left of the method's heading. For example:

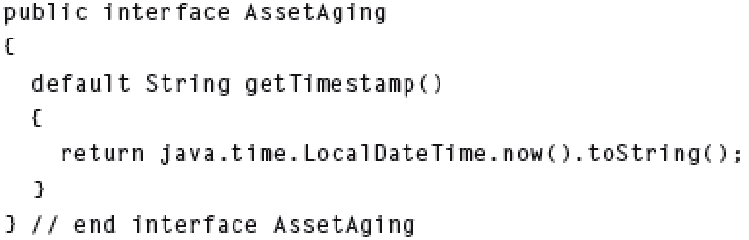
```
public interface AssetAging
{
  static void displayTimestamp()
  {
    System.out.println("Asset Aging account current time: " +
      java.time.LocalDateTime.now().toString());
  }
} // end interface AssetAging
```

Page 672

If a class's two interfaces implement

the same static method (same signature), there's no binding confusion, because to call an interface's static method, the method name must be prefaced with the interface in which the method is defined. For example, to call the above displayTimestamp method, you'd use AssetAging.displayTimestamp();.

A default method is a method that's defined within an interface with the default modifier at the left of the method's heading. For example:

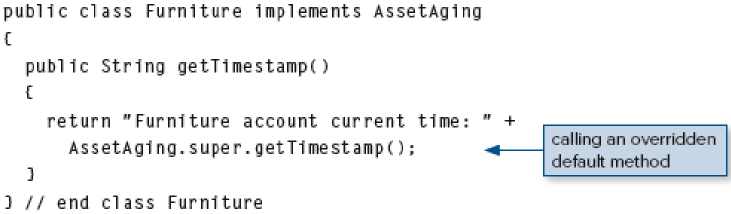


```
public interface AssetAging
{
  default String getTimestamp()
  {
    return java.time.LocalDateTime.now().toString();
  }
} // end interface AssetAging
```

Like a static method, a default method is only allowed to access the interface's static named constants, not any of the fields in an object whose class implements the interface. So how is an interface default method different from a static method? Static methods cannot be overridden; that rule applies to a subclass that extends a superclass as well as a class that implements an interface. On the other hand, a default method can be overridden. The need to override a default method should become apparent after we explain the raison d'être for default methods being introduced to interfaces.

With large real-world programs, using interfaces for lots of different classes is common. And having to update such programs is common. When the updates involve adding methods to an interface, in the old days, that required adding method definitions for all the classes that implemented the interface, even for the classes where the new methods were unnecessary. What a pain. With the advent of default methods, if a class that uses the updated interface doesn't need the new methods, no problem; the class is not required to provide definitions for those methods. Thus, backward compatibility is preserved for a program's existing classes.

Here's an example of a class implementing the AssetAging interface where the class overrides the interface's getTimestamp default method with its own getTimestamp method:



```
public class Furniture implements AssetAging
{
  public String getTimestamp()
  {
    return "Furniture account current time: " +
      AssetAging.super.getTimestamp();
  }
} // end class Furniture
```

calling an overridden default method

Suppose there's another interface that has its own version of the getTimestamp default method. If a class implements both interfaces, then an overriding method is no longer just an option; it's a requirement. Why would the compiler require the class to provide its own overriding getTimestamp method when there are two interfaces with the getTimestamp default method? Because otherwise, if an instance (object) of the class calls the getTimestamp method, the JVM would be confused as to which of the two interface default methods to bind to and execute. In the prior code fragment, note how the overriding getTimestamp method calls the interface's default method by prefacing the method name with AssetAging.super. You must include the interface's name, AssetAging, as part of the call. The interface's name is necessary to avoid an issue with multiple inheritance ambiguity.

Using an Interface to Store Universal Constants

So far, you've seen how an interface can serve as a contract for a class to ensure that it implements certain methods. You've seen how an interface can help with multiple inheritance. Now let's examine how to take advantage of all the named constants that an interface defines. Putting common named constants into an interface and then giving multiple classes access to those named constants by having them implement that interface is a handy way to provide easy access to a large set of common physical constants and/or empirical factors or constants. You avoid duplicate definitions of the named constants, and to access them you don’t have to worry about prefacing them with the class name. In principle, you could use an inheritance hierarchy to provide direct access to common named constants, but that would be bad practice because it would waste your one inheritance opportunity on nothing more than a bunch of named constants. If you use an interface to do this, you’re still free to use inheritance and/or additional interfaces for other purposes.

14.10 The protected Access Modifier

So far, we've discussed only two modes of accessibility for a class's members—public and private; public members can be accessed from anywhere; private members can be accessed only from inside the members' class. There is another access modifier that is a limited form of the public access modifier—the protected access modifier. It specifies an accessibility that's between public and private. Members that are protected can be accessed only from within the same package5 or from within the member's inheritance hierarchy. As you might recall, an inheritance hierarchy is the class in which the member resides, plus all its descendant classes.

When should you use the protected modifier? The general rule is that you should use it when you want easy access to a member, but you don't want to advertise it to the general public. In other words, you want it to have more exposure than a private member, but less exposure than a public member.6 Hmmm . . . that's still kind of vague. Let's elaborate with an example.

The Payroll Program with a protected Method

Suppose you want to enhance the Payroll program to include calculation of FICA taxes (FICA stands for Federal Insurance Contribution Act, and it funds the U.S. Social Security program). This tax calculation is best done in a separate method. Where should that method go? The only time this calculation will be done is when employees are paid. So, logically, it's a helper method called by the getpay method.

Where is the getPay method? It’s a polymorphic method that has an overriding redefinition in all classes that directly and indirectly extend the Employee class—Commissioned, Salaried, Hourly, and SalariedAndCommissioned. But hey! This set of classes, together with the Employee class itself, make up the Employee hierarchical tree. So instead of repeating the definition of the FICA calculation in all classes that have a getPay method, it’s more logical and more efficient to put this common FICA calculation in the tree’s root class, Employee, and make it protected.

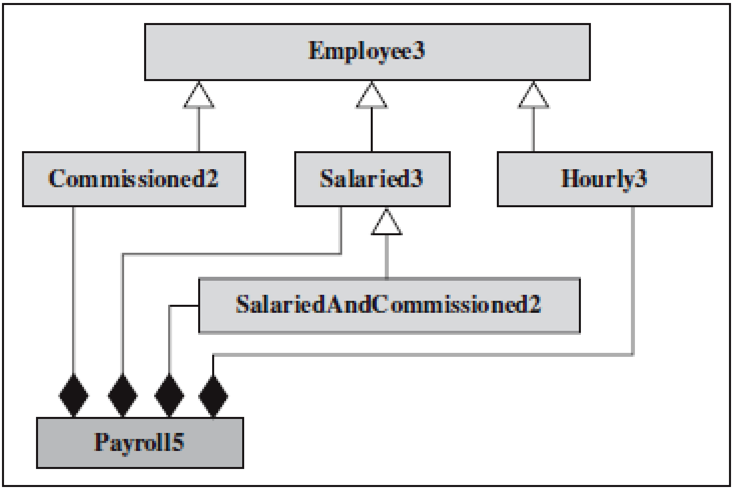

Employee3
Commissioned2
Salaried3
Hourly3
SalariedAndCommissioned2
Payroll5

Figure 14.19 Abbreviated class diagram for an enhanced Payroll program

To avoid trampling on previous versions of the program, we use new class names in our new FICA enhanced Payroll program. See Figure 14.19. It shows the program's UML diagram with the new class names—Payroll5, Employee3, Commissioned2, Salaried3, Hourly3, and SalariedAndCommissioned2.

Figure 14.20 shows the definition of Employee3, which implements a Commission2 interface. This interface is similar to the Commission interface in Figure 14.14. However, in accordance with the second AssetAging example at the end of the previous section, instead of:

```
void addSales(double sales);
```

the Commission2 interface uses:

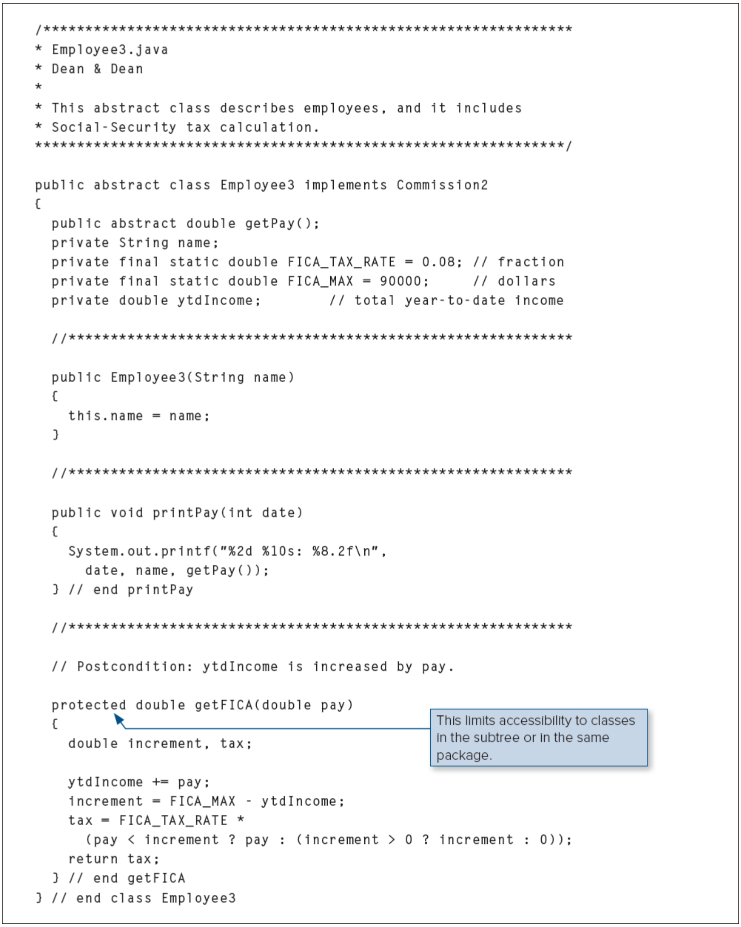
```
default void addSales(double sales) {}
```

The Employee3 class includes an additional common helper method, getFICA. Employee3 also includes some named constants used in the FICA calculation. The details of this calculation are not relevant to the present discussion, so to save space, we are implementing it in a fairly cryptic form using the conditional operator. What this little getFICA method does is a reasonable representation of what actually happens to people's paychecks. So if you're curious, you might want to expand the cryptic code into a more readable form. (An end-of-chapter exercise asks you to do this.)

Each of the polymorphic getPay methods includes a call to this new getFICA method. The code for this call is essentially the same in each of the getPay methods, so we'll show it just once, in the Salaried3 class in Figure 14.21.

For the most part, the SalariedAndCommissioned2 class that extends the Salaried3 class is like what's shown in Figure 14.16, with appropriate changes in the version numbers at the ends of the class names. However, in the getPay method, we cannot use super.getPay() to access salary in the Salaried3 class, because the FICA tax makes the value returned by Salaried3's getPay method different from the value of salary.

With the FICA tax, there must be another way to access salary. Although Salaried3 could include a getSalary accessor method, the code would be simpler if salary were public. But would you want everybody's salary to be public? Probably not. The most appropriate thing to do here is to elevate the accessibility of the salary variable in the Salaried3 class from private to protected. This gives descendant classes direct access to the salary variable, but it does not expose it as much as a public modifier would. Because the SalariedAndCommissioned2 class extends Salaried3, if there is a protected modifier on the salary variable in Salaried3, you can define the getPay method in the SalariedAndCommissioned2 class like this:

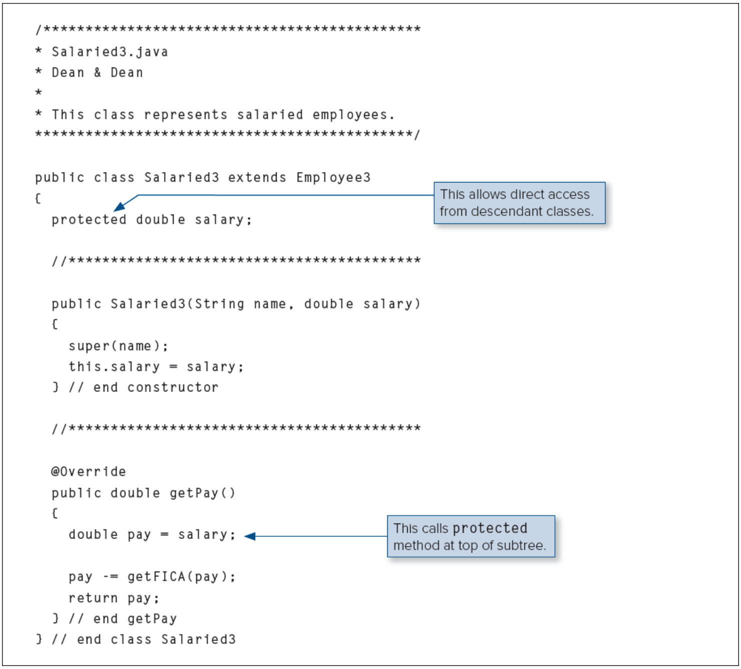


```
/***************************************************************
* Employee3.java
* Dean & Dean
*
* This abstract class describes employees, and it includes
* Social-Security tax calculation.
***************************************************************/

public abstract class Employee3 implements Commission2
{
  public abstract double getPay();
  private String name;
  private final static double FICA_TAX_RATE = 0.08; // fraction
  private final static double FICA_MAX = 90000;     // dollars
  private double ytdIncome;        // total year-to-date income

  //************************************************************

  public Employee3(String name)
  {
    this.name = name;
  }

  //************************************************************

  public void printPay(int date)
  {
    System.out.printf("%2d %10s: %8.2f\n",
      date, name, getPay());
  } // end printPay

  //************************************************************

  // Postcondition: ytdIncome is increased by pay.

  protected double getFICA(double pay)
  {
    double increment, tax;

    ytdIncome += pay;
    increment = FICA_MAX - ytdIncome;
    tax = FICA_TAX_RATE *
      (pay < increment ? pay : (increment > 0 ? increment : 0));
    return tax;
  } // end getFICA
} // end class Employee3
```

This limits accessibility to classes in the subtree or in the same package.

Figure 14.20 Employee3 class, which includes the protected getFICA method

Page 676

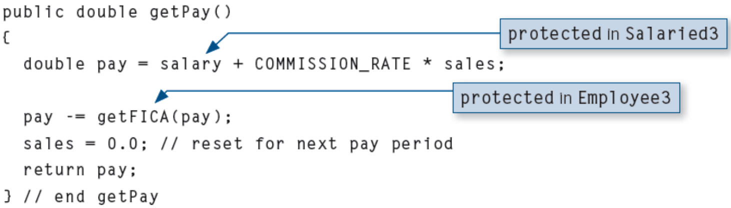


```
/*************************************************
* Salaried3.java
* Dean & Dean
*
* This class represents salaried employees.
************************************************/

public class Salaried3 extends Employee3
{
  protected double salary;

  //*******************************************

  public Salaried3(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //*******************************************

  @Override
  public double getPay()
  {
    double pay = salary;

    pay -= getFICA(pay);
    return pay;
  } // end getPay
} // end class Salaried3
```

This allows direct access from descendant classes.

This calls `protected` method at top of subtree.

Figure 14.21 Enhanced version of Salaried class that includes tax deduction



```
public double getPay()
{
  double pay = salary + COMMISSION_RATE * sales;

  pay -= getFICA(pay);
  sales = 0.0; // reset for next pay period
  return pay;
} // end getPay
```

`protected` in `Salaried3`

`protected` in `Employee3`

So there you have it. Polymorphism enables you to put heterogeneous objects into generic arrays whose type is either a class the objects' classes descend from or an interface the objects' classes implement. Then you can cast array elements into subclass or interface types, so the array elements can make method calls that are specific to their subclass or interface. The JVM finds the method that best matches the calling object and executes that method. The protected modifier allows direct access to variables and methods from anywhere in the protected member's subtree.

14.11 GUI Track: Three-Dimensional Graphics (Optional)

This section's examples will illustrate use of JavaFX's Shape3D classes with a PerspectiveCamera. In the first example the camera is in its default position in front of the center of the scene, and it looks in the positive z-direction, which is directly into the screen. The camera's distance from the screen (in the negative z-direction) just enables a 30-degree lens to see the full height of the center of the scene. The first example's lighting is the default lighting, which is a point source located at the camera's position. The second example illustrates a combination of Cylinder, Sphere, and Box objects viewed by an off-center camera and lighted by a point source at a different location and also by a diffuse light source. In the second example, user inputs move the camera to view the scene's objects from different directions and distances. Since a PerspectiveCamera makes more distant objects look smaller, it provides a satisfying sense of depth perception.

Cylinder Example

Our first example is a relatively simple program that presents a 3D image of a single cylinder oriented in accordance with user input. Figure 14.22 shows a typical output.

Figure 14.23a shows the first part of the program that generates the display shown in Figure 14.22 when user input specifies 120-degree rotation in the plane of the screen and 45-degree rotation out of this plane. Although the program itself is relatively simple, it requires many imports. If you have looked at previous end-of-chapter optional GUI sections, you will recognize Application, Stage, and Scene. The optional GUI sections at the ends of Chapter 5 and the Interlude also used Group and Color. PerspectiveCamera, PhongMaterial, Cylinder, Translate, and Rotate have not appeared previously in this book. PerspectiveCamera, Translate, and Rotate became available in JavaFX 2.0. PhongMaterial and Cylinder became available in JavaFX 8.0.

RotatedCylinder's start method

instantiates a zero-parameter PerspectiveCamera object. With this default camera, user alteration of a displayed window's size moves the camera to keep it centered in the window. This does not change the distance of the camera from the plane of the screen, so it does not alter the apparent size of scene contents. But it does alter the angles at which the camera views scene contents and the amount of contents it can see. Notice that the code assigns the ProspectiveCamera object directly to the Scene object. This puts the camera outside the Group object that will contain the scene's contents.

Figure 14.23b shows the createContents method, which creates and manipulates a Cylinder object with 100-pixel radius and 400-pixel height. The colored *Phong* material "reflects light in terms of a diffuse and specular component together with an ambient and a self-illumination term."7

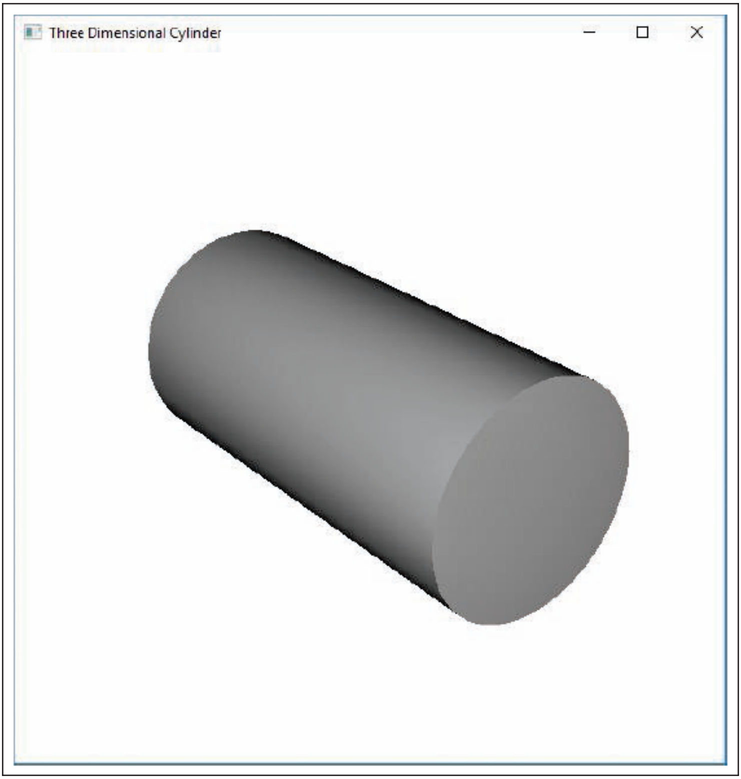


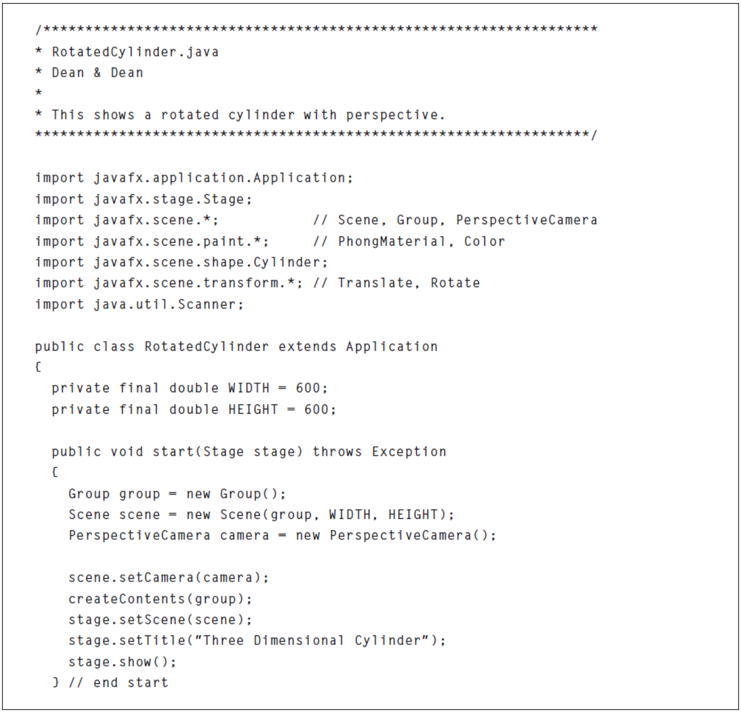
Figure 14.22 Light gray solid cylinder as seen by a camera with a 30-degree field of view. From an original orientation with axis in the y-direction, user input rotates the cylinder 120 degrees in the plane of the screen and then 45 degrees out of this plane.

©JavaFX

Initially, the cylinder's x-axis runs

horizontally with positive to the right, and its y-axis runs vertically with positive downward. Thus, initial x- and y-axes are in the plane of the screen. Initially, the cylinder's z-axis runs perpendicular to the plane of the screen with positive into the screen.

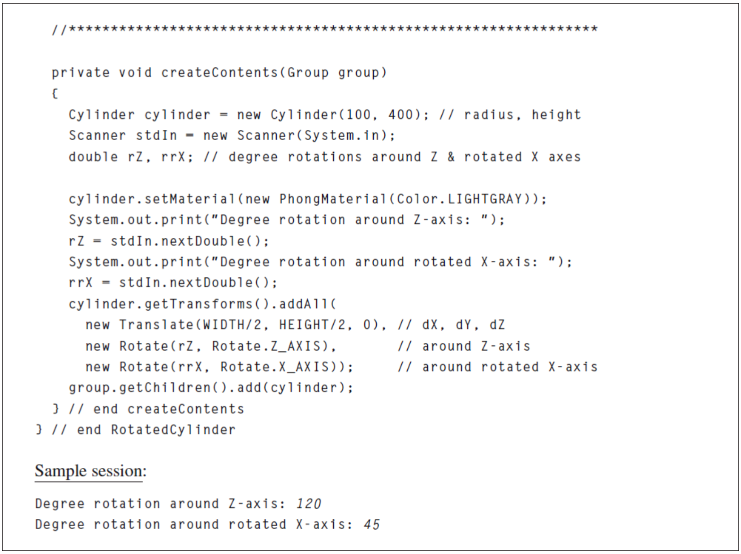
When you create a cylinder, the cylinder's axis is parallel to y, and the center of one side is at the upper left corner of the scene. Thus, if there were no transforms, what you would see would be the lower right quarter of the side of an upright cylinder in the upper left corner of the scene. Three quarters of the image would out of the picture, above and off to the left.



```
/*******************************************************************
* RotatedCylinder.java
* Dean & Dean
*
* This shows a rotated cylinder with perspective.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;           // Scene, Group, PerspectiveCamera
import javafx.scene.paint.*;     // PhongMaterial, Color
import javafx.scene.shape.Cylinder;
import javafx.scene.transform.*; // Translate, Rotate
import java.util.Scanner;

public class RotatedCylinder extends Application
{
  private final double WIDTH = 600;
  private final double HEIGHT = 600;

  public void start(Stage stage) throws Exception
  {
    Group group = new Group();
    Scene scene = new Scene(group, WIDTH, HEIGHT);
    PerspectiveCamera camera = new PerspectiveCamera();

    scene.setCamera(camera);
    createContents(group);
    stage.setScene(scene);
    stage.setTitle(”Three Dimensional Cylinder”);
    stage.show();
  } // end start
```

Figure 14.23a RotatedCylinder class—part A

Figure 14.23b’s first cylinder transform moves the cylinder from the scene’s upper left corner to the scene’s center. The second cylinder transform rotates the cylinder in the clockwise direction around the z-axis, that is, in the x-y plane, which is the plane of the screen. This rotation automatically rotates the cylinder’s x- and y-axes by the same amount. The third cylinder transform rotates the cylinder around the cylinder's now-rotated x-axis, whose positive direction is now down and left in the plane of the screen. All rotations conform to the *right-hand rule*: When your right thumb points in the positive direction of the axis, that hand's partially curled fingers point in the positive direction of rotation.



```
    //***************************************************************

    private void createContents(Group group)
    {
      Cylinder cylinder = new Cylinder(100, 400); // radius, height
      Scanner stdIn = new Scanner(System.in);
      double rZ, rrX; // degree rotations around Z & rotated X axes

      cylinder.setMaterial(new PhongMaterial(Color.LIGHTGRAY));
      System.out.print("Degree rotation around Z-axis: ");
      rZ = stdIn.nextDouble();
      System.out.print("Degree rotation around rotated X-axis: ");
      rrX = stdIn.nextDouble();
      cylinder.getTransforms().addAll(
        new Translate(WIDTH/2, HEIGHT/2, 0), // dX, dY, dZ
        new Rotate(rZ, Rotate.Z_AXIS),       // around Z-axis
        new Rotate(rrX, Rotate.X_AXIS));     // around rotated X-axis
      group.getChildren().add(cylinder);
    } // end createContents
  } // end RotatedCylinder
```

Sample session:

```
Degree rotation around Z-axis: 120
Degree rotation around rotated X-axis: 45
```

Figure 14.23b RotatedCylinder class—part B

In Figure 14.23b's sample session, the first user input of 120 makes the second cylinder transform rotate the cylinder 120 clockwise in the plane of the screen. The second user input of 45 makes the third cylinder transform rotate the upper left end of the cylinder into the screen. After these two rotations, the end we see on the right side of the scene is the top of the unrotated cylinder. We could get the same visual effect with the alternate user entries, −60 and −45. But then, the end we would see on the right side of the scene would be the bottom of the unrotated cylinder.

Landscape Example

If we override the default PerspectiveCamera by constructing it with a true argument, we can vary camera and light source positions. Figure 14.24 shows a sequence of views seen by the driver of an automobile traveling down a country road. In addition to the road there is a grass-covered plane and a signpost. To generate the six images in Figure 14.24, a user executes the subsequently described Signpost program with inputs, $+20$, $+10$, $+3$, $+1$, -10, and -50, respectively.

In addition to perspective, this example illustrates changing viewpoint and lighting that comes from a point different from the viewpoint. Although the shading on our signpost's surface is realistic, notice that the signpost does not cast a shadow on the grass behind it. The Java 9 API does not account for shading of one object by another object. Also, there is no straightforward way to put text on one of these objects. That's probably OK here because there's not much our sign could say about the surrounding landscape.

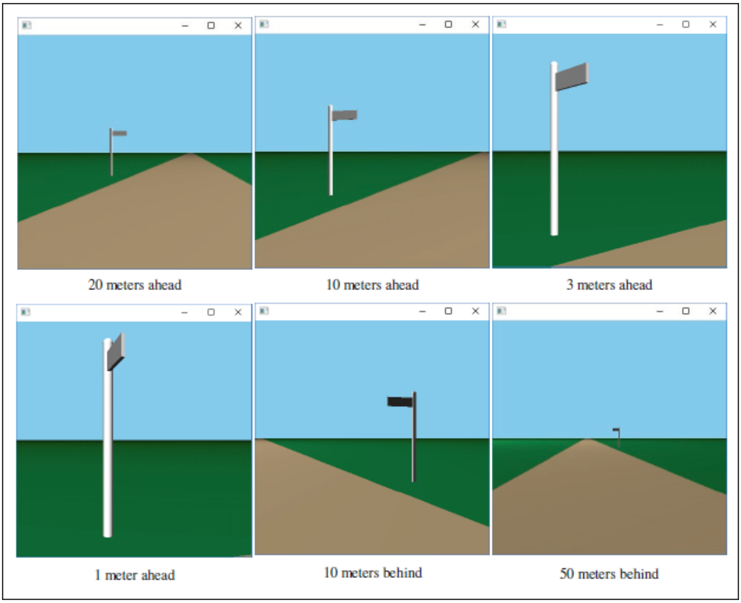
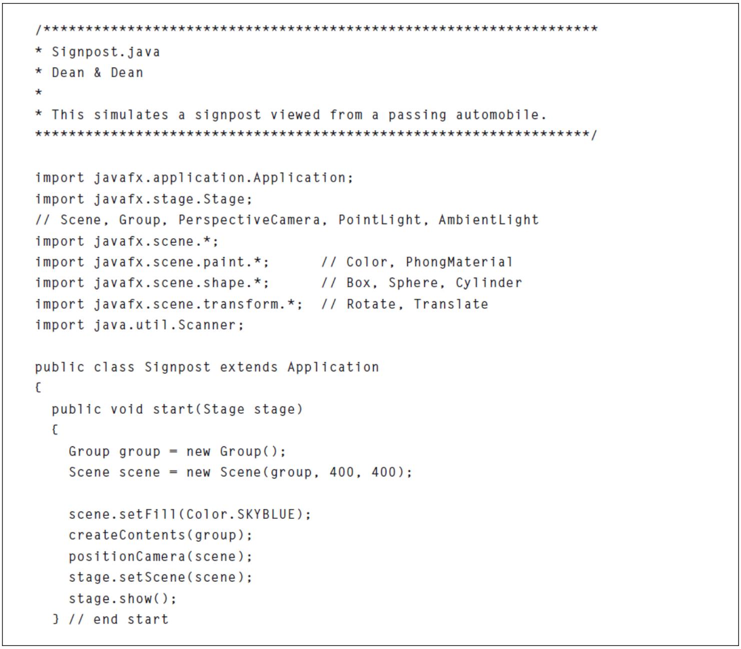

20 meters ahead
10 meters ahead
3 meters ahead
1 meter ahead
10 meters behind
50 meters behind

Figure 14.24 Six views of a 10-foot-high sign beside a 20-foot-wide country road with a 40-degree field of view. The sun is above and behind the approaching traveler's right shoulder, making the traveler see the illuminated side of the sign when approaching and the shaded side when leaving.

Figure 14.25a shows the first part of the program that generates the displays in Figure 14.24. This program's import statements provide access to a large amount of prewritten

API software. The start method declares and initializes a Group object called group and puts it into a scene with arbitrary width and height values, which have no necessary relation to scene-component dimensions. Scene's setFill method provides sky coloring. The createContents method adds scene components and light sources. The positionCamera method creates a custom PerspectiveCamera.

Figure 14.25b shows the createContents method. The first declaration creates a Box for the grass-covered plane. This box extends 40,000 pixels in the x- and z-directions and has zero thickness in the y-direction. Using a scaling factor of one pixel per inch and noting that there are 39.4 inches per meter, our grass-covered plane extends approximately 1 km in the x-direction and 1 km in the z-direction. The second declaration creates another Box to represent the road. It extends 40,000 pixels in the x-direction, 6 pixels in the y-direction, and 240 pixels in the z-direction. This second box represents a section of road that is 6 inches thick, 20 feet wide, and 1 km long.

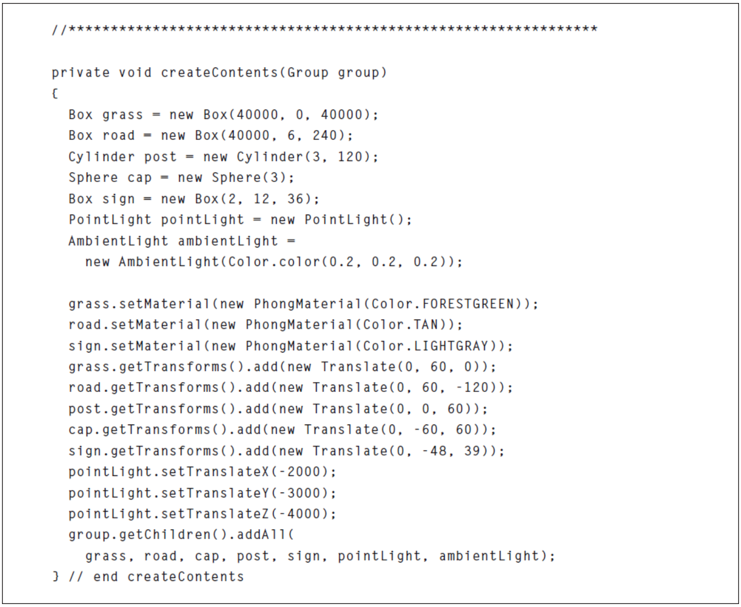


```
/*********************************************************************
* Signpost.java
* Dean & Dean
*
* This simulates a signpost viewed from a passing automobile.
*********************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
// Scene, Group, PerspectiveCamera, PointLight, AmbientLight
import javafx.scene.*;
import javafx.scene.paint.*;      // Color, PhongMaterial
import javafx.scene.shape.*;      // Box, Sphere, Cylinder
import javafx.scene.transform.*;  // Rotate, Translate
import java.util.Scanner;

public class Signpost extends Application
{
  public void start(Stage stage)
  {
    Group group = new Group();
    Scene scene = new Scene(group, 400, 400);

    scene.setFill(Color.SKYBLUE);
    createContents(group);
    positionCamera(scene);
    stage.setScene(scene);
    stage.show();
  } // end start
```

Figure 14.25a Signpost class—part A

The third, fourth, and fifth declarations create signpost components. The Cylinder with 3-pixel radius and 120-pixel height represents the 6-inch-diameter and 10-foot-high post. The Sphere with 3-pixel radius represents a cap for the top of the post. The x = 2, y = 12, z = 36 Box represents a 2-inch-thick, 12-inch-high, and 36-inch-long sign extending from one side of the post. The PointLight and AmbientLight objects represent point and diffuse light sources, respectively. The ambient light's subdued color adds a little light to the shaded side of the post and the bottom and back sides of the sign, as you can see in the last three images in Figure 14.24.

The first three statements after the declarations use the PhongMaterial class to color the grass green, the road tan, and the sign light gray. Those are straightforward operations. The next five statements employ getTransforms().add(new Translate(*dx*, *dy*, *dz*)) method calls to translate the object seen by the camera relative to the camera's point of focus, which is at the exact center of each of the six images in Figure 14.24.



```
//*****************************************************************

private void createContents(Group group)
{
  Box grass = new Box(40000, 0, 40000);
  Box road = new Box(40000, 6, 240);
  Cylinder post = new Cylinder(3, 120);
  Sphere cap = new Sphere(3);
  Box sign = new Box(2, 12, 36);
  PointLight pointLight = new PointLight();
  AmbientLight ambientLight =
    new AmbientLight(Color.color(0.2, 0.2, 0.2));

  grass.setMaterial(new PhongMaterial(Color.FORESTGREEN));
  road.setMaterial(new PhongMaterial(Color.TAN));
  sign.setMaterial(new PhongMaterial(Color.LIGHTGRAY));
  grass.getTransforms().add(new Translate(0, 60, 0));
  road.getTransforms().add(new Translate(0, 60, -120));
  post.getTransforms().add(new Translate(0, 0, 60));
  cap.getTransforms().add(new Translate(0, -60, 60));
  sign.getTransforms().add(new Translate(0, -48, 39));
  pointLight.setTranslateX(-2000);
  pointLight.setTranslateY(-3000);
  pointLight.setTranslateZ(-4000);
  group.getChildren().addAll(
    grass, road, cap, post, sign, pointLight, ambientLight);
} // end createContents
```

Figure 14.25b Signpost class—part B

Page 683

The grass transform moves the plane of grass down (in the positive y-direction) by 60 pixels (5 feet) to position it even with the bottom of the 120-pixel-high (10-foot-high) post. At large positive or negative values of x or high z, this translated grass plane's horizon crosses behind the camera's point of focus.

The road transform moves the center of the road 60 pixels down (in the positive y-direction) to align it with the grass plane. This transform also moves the road out of the plane of the screen (in the negative z-direction) by half the road's width. This makes the far edge of the road pass five feet below the camera's point of focus.

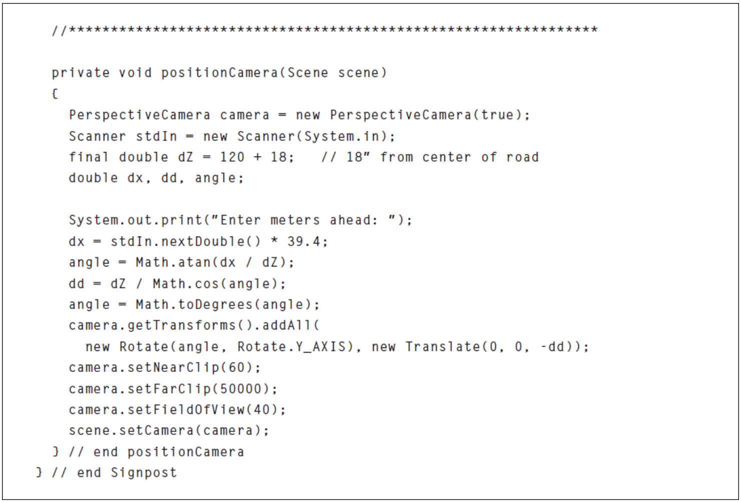
The post transform moves the center of the post 60 pixels in the positive z-direction. This puts the post 5 feet behind the point of focus and the far edge of the road.

The cap transform moves the sphere up to the top of the post. The sign transform moves the sign 48 inches in the negative y-direction and in 39 inches in the positive z-direction. This puts the top of the 12-inch-high by 36-inch-long sign 6 inches below the top of the post and brings the far end of that sign into contact with the near side of the 6-inch-diameter post.

The three pointLight translations position the scene's light source 2000 pixels, 3000 pixels, and 4000 pixels in the negative x-, y-, and z-directions, respectively. This puts the light above the right shoulder of a traveler approaching the sign. At 1 inch per pixel, this light source is about 450 feet away from the camera's center of focus. That's somewhat shorter than the distance to the sun but, oh well, this scene is already spooky because its objects don't cast shadows.

The final createContents statement adds all the previous objects to the passed-in group container. The order of the first five addAll arguments is important, because each additional item overlays all previous items. The road and all signpost components overlay the grass. Because the post overlays the cap, it occludes the lower half of cap's spherical shape, making it appear hemispherical. And because sign overlays post, it occludes whatever post material is behind it.8

Figure 14.25c shows the positionCamera method. The first declaration constructs a PerspectiveCamera object with a true argument. This true argument implements a special functionality called “fixed eye at camera zero,” which we’ll describe shortly. The scanned user input is the positive x-distance in meters from the viewer to the location of the signpost. Using one pixel per inch, the code converts this input to pixels. Then it computes the angle from the negative z-direction to the current camera position and the pixel distance from the point of focus to the current camera position.



```
  //***************************************************************

  private void positionCamera(Scene scene)
  {
    PerspectiveCamera camera = new PerspectiveCamera(true);
    Scanner stdIn = new Scanner(System.in);
    final double dZ = 120 + 18;   // 18" from center of road
    double dx, dd, angle;

    System.out.print("Enter meters ahead: ");
    dx = stdIn.nextDouble() * 39.4;
    angle = Math.atan(dx / dZ);
    dd = dZ / Math.cos(angle);
    angle = Math.toDegrees(angle);
    camera.getTransforms().addAll(
      new Rotate(angle, Rotate.Y_AXIS), new Translate(0, 0, -dd));
    camera.setNearClip(60);
    camera.setFarClip(50000);
    camera.setFieldOfView(40);
    scene.setCamera(camera);
  } // end positionCamera
} // end Signpost
```

Figure 14.25c Signpost class—part C

With PerspectiveCamera(true) the programmer needs to set camera position with camera transforms and identify near and far clip distances. The near clip should be closer to the camera than any viewed object and the far clip should be farther from the camera than any viewed object.

In a PerspectiveCamera(true) implementation, the initial point of focus is in the upper left corner, where all newly created objects initially appear. This eliminates the need for explicit object translations to the center of the screen, as in the previous example. So how did the point of focus end up in the center of the images in Figure 14.24? The true argument automatically performs this translation for the whole group.

The true argument also does something else. It scales all pixels in the scene by an internally computed *scaling factor*:

scaling-factor = *height* / (2 * *distance* * Math.tan(*field-of-view* / 2))

where:

height is the program's explicit height specification supplied in the scene constructor;

distance is distance from the specified camera position to the point of focus;

field-of-view is either the default (30-degree) or an explicitly specified value.

In the current example, *height* = 400. If the user enters 0 in response to the

"Enter meters ahead:" prompt, *distance* = 138. The *field-of-view* explicitly specified in the positionCamera method's next-to-last statement is 40. With these values, *scaling-factor* = 4.0.

With this scaling factor, the pixel height of the post increases to 4 * 120 = 480. If the post were at the center of focus, its height would exceed the window's height. But the post's 60-pixel offset in the +z-direction makes it appear smaller by a factor of 138 / (138 + 60) = 0.7 and 0.7 * 480 = 335, which is 84% of the 400-pixel window height and close to what you see in the fourth image in Figure 14.24. If the program or the user increases window height, everything in the scene scales up in the same proportion. This enlargement of all the scene's content pushes some content at the sides of the scene out of the field of view.

On the other hand, window width

changes leave the scaling factor unaltered. As in the previous example, width changes just change the horizontal extent of visible scenery. However, the true option preserves automatic centering, and view angles do not change as they do when window size changes with a zero-parameter ProspectiveCamera.

Summary

- The Object class is the ancestor of all other classes.
- To avoid using the Object class's equals method, for each of your classes, you should define an equals method that compares instance variable values.
- To avoid the Object class's mysterious response to a toString method call, for each of your classes, you should define a

toString method that outputs a string concatenation of instance variable values.

- At compile time, the compiler confirms that a reference variable’s class is able to handle each of the reference variable’s method calls in some way. At runtime, the JVM looks at the particular type of the object referred to by the reference variable to determine which one of several alternative polymorphic methods should actually be called, and it binds the object to that method.

- The instanceof operator enables you to determine explicitly whether the object referred to by a reference variable is an instance of a particular class or descended from that class.

- You can always assign an object to a more generic reference variable, because the object's methods include methods inherited from the reference variable's class.

- You can safely cast a more generic reference into a more specific type only if you know the actual object referred to will be as specific as or more specific than the cast.

- You can implement polymorphism in an array of heterogeneous objects by declaring the array elements to be instances of a common inheritance ancestor. To satisfy the compiler, you can write a dummy method in that ancestor class and override it in all classes instantiated in the array. Or you can declare the method in an abstract ancestor class and then define overriding methods in all descendant classes instantiated in the array. Or you can declare the method in an interface and implement that interface in all classes instantiated in the array.

- A class can extend one inherited superclass and/or implement any number of interfaces.

- An interface provides simple access to common constants.

- The protected access modifier provides direct access to members of classes in the same package or in the inheritance subtree whose root is the class in which the protected member is declared.

· With the aid of explicit trigonometric calculations, you can use Java API classes to draw what appear to be three-dimensional objects.

Review Questions

§14.2 The Object Class and Automatic Type Promotion

1. If you want a class you define to inherit methods from the Object class, you must append the suffix extends Object to your class's heading. (T / F)

§14.3 The equals Method

2. When used to compare reference variables, the == operator works the same as the Object class's equals method. (T / F)

3. What does the equals method defined in the String class compare?

§14.4 The toString Method

4. What is returned by the Object

class’s toString method?

5. What’s wrong with replacing the println statement in Figure 14.2’s main method with these two statements?

```
String description = car.toString();
System.out.println(description);
```

6. The return type of an overriding method must be the same as the return type of the overridden method. (T / F)

§14.5 Polymorphism and Dynamic Binding

7. In Java, polymorphic method calls are bound to method definitions at compile time (not runtime). (T / F)

§14.6 Assignments When the Two Sides’ Classes Are Different

8. Assume one reference variable's class is descended from another reference variable's class. To be able to assign one reference variable to the other one (without using a cast operator), the left-side variable's class must be a(n) __________ of the right-side reference variable's class.

§14.7 Polymorphism with Arrays

Page 687

9. A given array may contain different types of elements. (T / F)

§14.8 abstract Methods and Classes

10. What are the syntax features of an abstract method?

11. Any class that contains an abstract method must be declared to be an abstract class. (T / F)

12. You cannot instantiate an

abstract class. (T / F)

§14.9 Interfaces

13. You can use an interface to provide direct access to a common set of constants from many different classes. (T / F)

14. You can declare reference variables to have an interface type and use them just as you would use reference variables declared to be the type of a class in an inheritance hierarchy. (T / F)

§14.10 The protected Access Modifier

15. Describe the access provided by the protected modifier.

16. It's illegal to use private for any method that overrides an abstract method. (T / F)

Exercises

1. [after §14.3] Given a Car class with the following instance variables:

private String make;

private String model;

private int year;

private String color;

write an equals method for customers who don't care about color.

2. [after §14.4] Assume that this chapter's Car class is instantiated with a statement like this:

Car car = new Car("Mazda", 2018, "green");

Enhance this Car class by providing an additional method that responds to this subsequent print statement:

System.out.println(car);

with output like this:

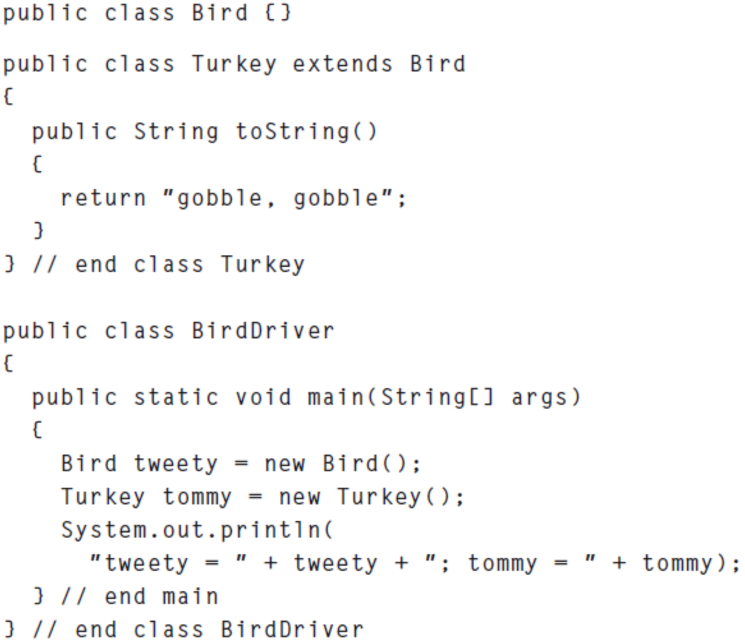
```
green 2018 Mazda
```

3. [after §14.4] Assume you have a programmer-defined class named Person. Assume this class includes a name instance variable, initialized by the constructor's name parameter. Suppose a driver fills a people array with Person objects. Then the driver goes back through the array and uses System.out.println(people[i]); to print the people's names. Unfortunately, what the driver prints is something like this:

```
Person@65ae6ba4
Person@512ddf17
Person@2c13da15
Person@77556fd
```

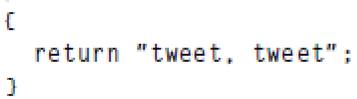
What is the problem and how can you fix it?

4. [after §14.4] What does the following program output? For each bird's output, describe how the output is generated (be specific).



```
public class Bird {}

public class Turkey extends Bird
{
  public String toString()
  {
    return "gobble, gobble";
  }
} // end class Turkey

public class BirdDriver
{
  public static void main(String[] args)
  {
    Bird tweety = new Bird();
    Turkey tommy = new Turkey();
    System.out.println(
      "tweety = " + tweety + "; tommy = " + tommy);
  } // end main
} // end class BirdDriver
```

5. [after §14.5] Suppose we add the following method to the previous exercise's Bird class:



```
public String toString()
{
  return "tweet, tweet";
}
```

What does the enhanced program output? Explain.

6. [after §14.5] Suppose you have a reference variable named thing, you're not sure what type of object it points to, and you want your program print out its type. The Object class (and therefore any class!) has another method, getClass, that returns a special object of type Class that contains information about the class of the object calling the getClass method. The Class class has a method called getName which returns the name of the class described by its calling object. Write a statement that prints the name of thing's class.

7. [after §14.6] Given superclass Pet and subclass Cat, identify and explain all compilation errors in the following code fragment.

```
Pet pet;
Cat mrWhiskers, fluffy = new Cat();

mrWhiskers = new Pet();
pet = fluffy;
```

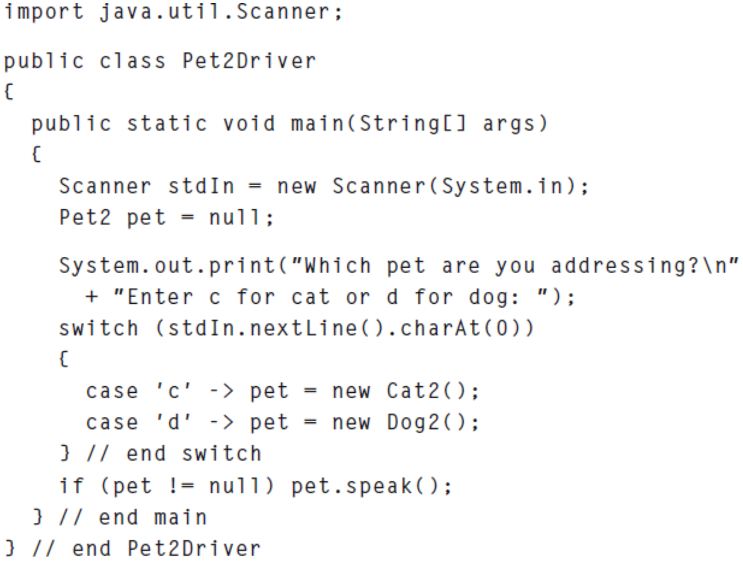
8. [after §14.7] Assume you have

a superclass Pet and subclasses Cat and Dog. In the following code fragment, the second and third lines generate compilation errors. Provide corrected versions of those two lines. Preserve the spirit of the original code. For example, the bottom line should still assign the second pets element into the barky variable.

```
Pet[] pets = {new Cat(), new Dog()};
Cat tiger = pets[0];
Dog barky = pets[1];
```

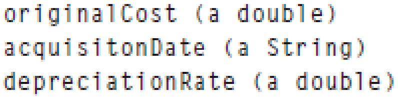
9. [after §14.8] If a superclass is abstract, every subclass must override all of the superclass's methods (T / F).

10. [after §14.8] Given the Pet2Driver class below, write an abstract Pet2 class that contains just one item—an abstract declaration for a speak method. Also write Dog2 and Cat2 classes that extend Pet2, so that when you execute Pet2Driver and input ‘c’ or ‘d’, the program prints “Meow! Meow!” or “Woof! Woof” but if you input anything else, it does not print anything.



```
import java.util.Scanner;

public class Pet2Driver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Pet2 pet = null;

    System.out.print("Which pet are you addressing?\n"
      + "Enter c for cat or d for dog: ");
    switch (stdIn.nextLine().charAt(0))
    {
      case 'c' -> pet = new Cat2();
      case 'd' -> pet = new Dog2();
    } // end switch
    if (pet != null) pet.speak();
  } // end main
} // end Pet2Driver
```

11. [after §14.9] Write an AssetAging interface that includes abstract accessor methods that retrieve the following instance variables’ values:



```
originalCost (a double)
acquisitonDate (a String)
depreciationRate (a double)
```

12. [after §14.9] In Exercise 10, above, indicate all changes you would make to Pet2, Cat2, and

Dog2 so that Pet2 is an interface rather than an abstract class. It should not be necessary to make any changes to Exercise 10's Pet2Driver class.

13. [after §14.10] Expand the cryptic code in the getFICA method of the Employee3 class in Figure 14.20 into "if else" statements so that the algorithm is easier to understand.

14. [after §14.7] (Case Study) This exercise refers to the different versions of the GridWorld program that have been looked at previously. Identify a GridWorld class that stores a two-dimensional array that can hold objects of any type. Provide the class name and the declaration for the array variable. List three types of objects that have been stored in the array.

15. [after §14.9] (Case Study)

This exercise refers to the different versions of the GridWorld program that have been looked at previously. Identify a GridWorld interface that is implemented by a class. Provide the method headings for each method that the interface specifies.

16. [after §14.9] (Case Study) Describe GridWorld’s Location class. What Java API interface does it implement? Identify its instance variables, summarize its static constants, and list its method headers.

Review Question Solutions

1. False. Every class is a descendant of the Object class, so specifying extends Object is not necessary. In fact, it is undesirable because it prevents the extension of some other class.

2. True.

3. The equals method defined in the String class compares a string's characters.

4. The Object class's toString method returns a string concatenation of these three text components:

- full classname
- @ character
- a hexadecimal hashcode value

5. Nothing. It's just a matter of style—whether you want more compactness or more self-documentation.

6. True.

7. False. At runtime. The JVM determines which method is called.

8. To be able to assign one reference variable to the other one (without using a cast operator), the left-side variable's class must be a(n) superclass/ ancestor of the right-side reference variable's class.

9. True, if each element's type is either the type defined in the array declaration or a descendant of that type (or conforms to the interface that defines the array's type—see Section 14.9).

10. The syntax features of an abstract method are:

- The method heading contains the abstract modifier.
- There is a semicolon at the end of the heading.
- There is no method body.

11. True.

12. True.

13. True

14. True.

15. It is legal to access a protected member:

- from within the same class as the protected member
- from within a class descended from the protected member
- from within the same package

16. True. An abstract method must be public or protected (it cannot be private). An overriding method must be no more restrictive than its overridden method. Therefore, if a method overrides an abstract method, it cannot be private.

---

[1] To get an idea of how common equals methods are, go to Oracle's Java API website and search for occurrences of

equals.

[2] Curious George is the main character in a series of books written by Margret and H. A. Rey. George is a curious monkey. Author John's daughter, Caiden, is a Curious George wannabe.

[3] Here's an example of how the last day in the current month can be found:

[4] See Appendix 7 for a complete UML diagram of the enhanced Payroll program developed in this subsection.

[5] If you want to learn more about packages and how to group your classes into a programmer-defined package, see Appendix 4.

[6] Because a protected member can be accessed from any class descended from the class that defines the protected member, anyone could extend the class that defines the protected member and thereby gain direct access to it. In other words, the

protected modifier doesn't actually provide much protection. If you're an outsider, stay away from someone else's protected members. Consider them to be nonstandard products that are not guaranteed.

[7] From the OpenJFX documentation, https://openjfx.io/javadoc/12/javafx.graphics/javafx/scene/paint/PhongMaterial.html. A Phong material is a smoothed rendition of differently oriented adjacent flat surfaces. The interpolation technique was developed at the University of Utah by Bui Tuong Phong.

[8] This works for the current program. However, if the program were able to position the camera on the positive-z side of the signpost, the relationship between sign and post would be distorted.

Objectives

- Understand what an exception is.
- Use `try` and `catch` blocks for numeric input validation.
- Understand how `catch` blocks catch an exception.
- Explain the difference between checked and unchecked

exceptions.

- Look up exception details on Oracle’s Java API website.
- Catch exceptions with the generic `Exception` class.
- Use the `getClass` and `getMessage` methods.
- Catch exceptions with multiple `catch` blocks and multiple-exception `catch` parameters.
- Understand exception messages.
- Propagate exceptions back to the calling module with the help of a `throws` clause.
- Learn how to close resources explicitly or close them implicitly in a try-with-resources block.

Outline

15.1 Introduction 692

15.2 Overview of Exceptions and Exception Messages 692

15.3 Using `try` and `catch` Blocks to Handle “Dangerous” Method Calls 693

15.4 Line Plot Example 695

15.5 `try` Block Details 699

15.6 Two Categories of Exceptions—Checked and Unchecked 700

15.7 Unchecked Exceptions 702

15.8 Checked Exceptions 705

15.9 Generic `catch` Block with `Exception` Class 708

15.10 Multiple `catch` Blocks and Multiple Exceptions per Block 712

15.11 Understanding Exception Messages 714

15.12 Using a `throws` Clause to Postpone the `catch` 718

15.13 Automatic Cleanup Using Try-With-Resources 720

15.14 GUI Track: Line Plot Example Revisited (Optional) 722

15.1 Introduction

As you know, programs sometimes generate errors. Compile-time errors deal with incorrect syntax, like forgetting parentheses around an `if` statement condition. Runtime errors deal with code that behaves inappropriately, like trying to divide by zero. In previous chapters, we fixed compile-time errors by correcting the erroneous syntax, and we fixed runtime errors by making code more robust. In this chapter, we deal with errors using a different technique—exception handling. We'll describe exceptions more formally later on, but for now, think of an exception as an error, or simply something that goes wrong with a program. Exception handling is an elegant way to deal with such problems.

We start this chapter by looking at a common problem—making sure that users enter a valid number when they are asked for a numeric input. You'll learn how to implement such input validation using `try` and `catch` blocks, two of the key exception handling constructs. There are different types of exceptions, and you'll learn how to deal with the different types appropriately. In the chapter's final section, you'll use exception handling as part of a GUI line-plot program.

📖 To understand this chapter, you need to be familiar with object-oriented programming, arrays, and inheritance basics. As such, you need to have read up through Chapter 13. This chapter does not depend on material covered in Chapter 14.

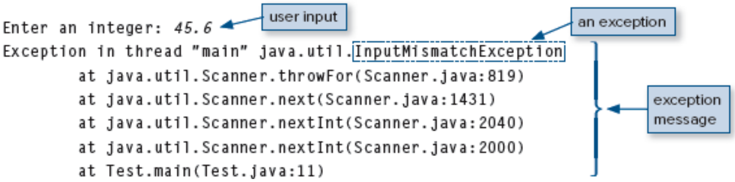
Different readers may want to read different parts of this chapter (Exception Handling) and the next chapter (Files, Buffers, Channels, and Paths). If you plan to read the next chapter, then you'll need to read this chapter in its entirety because the topic addressed in the next chapter, file manipulation, relies heavily on exception handling. On the other hand, if you plan to skip the next chapter and go directly to Chapters 17, 18, and 19 (GUI programming) or to Chapters S17 and S18 on the website (FX GUI programming), then you'll need to read only the first part of this chapter, Sections 15.1 through 15.7.

15.2 Overview of

Exceptions and Exception Messages

As defined by Oracle,[1] an *exception* is an event that disrupts the normal flow of instructions during the execution of a program. *Exception handling* is a technique for handling such exceptions gracefully.

⚠ The first exceptions we'll look at deal with invalid user input. Have you ever crashed a program (made it terminate ungracefully) due to invalid input? If a program calls the `Scanner` class's `nextInt` method and a user enters a noninteger, the JVM generates an exception, displays a big, ugly error message, and terminates the program. Here's a sample session that illustrates what we're talking about:



```
Enter an integer: 45.6
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:819)
        at java.util.Scanner.next(Scanner.java:1431)
        at java.util.Scanner.nextInt(Scanner.java:2040)
        at java.util.Scanner.nextInt(Scanner.java:2000)
        at Test.main(Test.java:11)
```

Note the `InputMismatchException` above. That's the type of exception that's generated when a user enters a noninteger in response to a `nextInt` method call. Note the *exception message*. Exception messages can be annoying, but they serve a useful purpose. They provide information about what's gone wrong. Toward the end of this chapter, we cover exception message details. But first, a more important issue—how to avoid getting ugly exception messages in the first place. Let us begin.

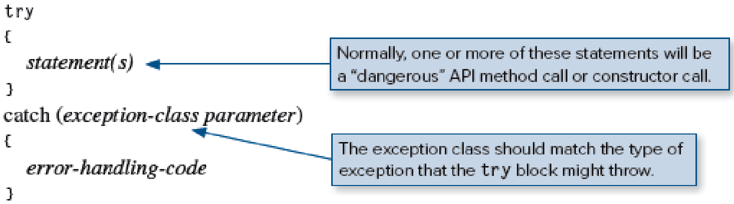
15.3 Using try and catch Blocks to Handle "Dangerous" Method Calls

Some method calls, like `nextInt`, are dangerous in that they can lead to exceptions, and exceptions can lead to program crashes. By the way, “dangerous” is not a standard exception handling term, but we’ll use it because it helps with explanations. In this section, we describe how to use `try` and `catch` blocks to fend off exception messages and program crashes. Use a `try` block to “try” out one or more dangerous method calls. If there’s a problem with the dangerous method call(s), the JVM jumps to a `catch` block and the JVM executes the `catch` block’s enclosed statements. Drawing an analogy, a `try` block is like a circus trapeze act. A trapeze act contains one or more dangerous stunts, like a triple flip or a triple twist. The dangerous stunts are like dangerous method calls. If something goes wrong with one of the stunts and an acrobat falls, there’s a net to catch the acrobat.

Likewise, if something goes wrong with one of the dangerous method calls, control passes to a `catch` block. If nothing goes wrong with the trapeze stunts, the net isn't used at all. Likewise, if nothing goes wrong with the dangerous method calls, the `catch` block isn't used at all.

Syntax and Semantics

Here's the syntax for `try` and `catch` blocks:



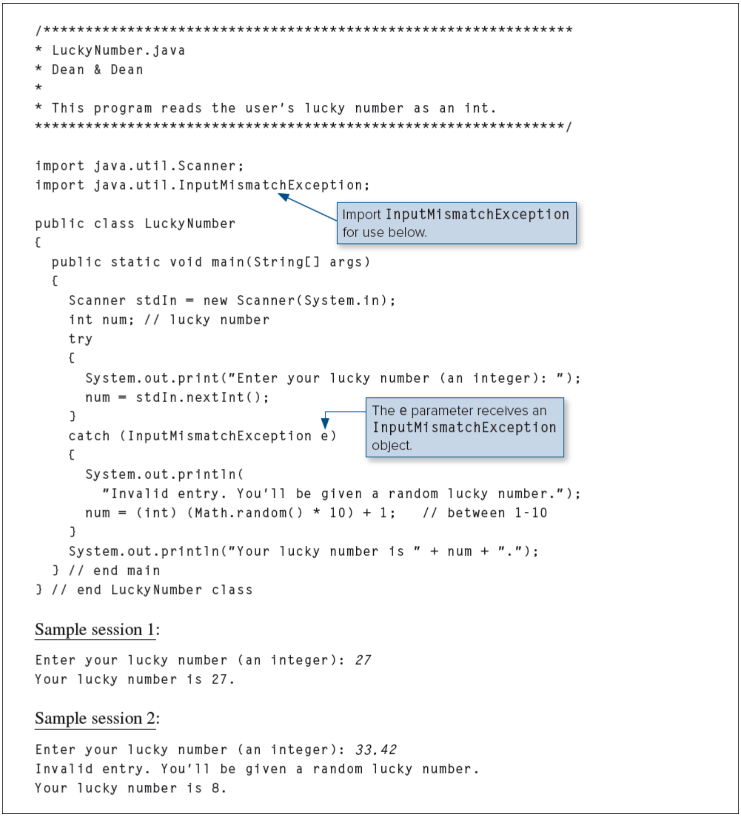
```
try
{
   statement(s)
}
catch (exception-class parameter)
{
   error-handling-code
}
```

Normally, one or more of these statements will be a "dangerous" API method call or constructor call.

The exception class should match the type of exception that the `try` block might throw.

As shown above, a `try` block and its associated `catch` block (or multiple `catch` blocks, which we'll address later) must be contiguous. You can put other statements before the `try` block or after the (last) `catch` block, but not between them. Note the parameter in the `catch` block's heading. We'll explain `catch` block parameters in the context of the following example program.

See Figure 15.1's LuckyNumber program. Note how the `try` and `catch` blocks follow the syntax pattern shown above. Within the `try` block, the `nextInt` method call tries to convert a user entry to an integer. For the conversion to work, the user entry must contain only digits and an optional preceding minus sign. If the user entry conforms to that format, the JVM assigns the user entry to the `num` variable, skips the `catch` block, and continues with the code below the `catch` block. If the user entry does not conform to that format, an exception occurs. If an exception occurs, the JVM immediately exits from the `try` block and instantiates an *exception object*—an object that contains information about the exception event.



```
/*******************************************************************
* LuckyNumber.java
* Dean & Dean
*
* This program reads the user’s lucky number as an int.
*******************************************************************/

import java.util.Scanner;
import java.util.InputMismatchException;

public class LuckyNumber
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int num; // lucky number
    try
    {
      System.out.print(”Enter your lucky number (an integer): ”);
      num = stdIn.nextInt();
    }
    catch (InputMismatchException e)
    {
      System.out.println(
        ”Invalid entry. You’ll be given a random lucky number.”);
      num = (int) (Math.random() * 10) + 1;   // between 1-10
    }
    System.out.println(”Your lucky number is ” + num + ”.”);
  } // end main
} // end LuckyNumber class
```

Import `InputMismatchException` for use below.

The `e` parameter receives an `InputMismatchException` object.

Sample session 1:

```
Enter your lucky number (an integer): 27
Your lucky number is 27.
```

Sample session 2:

```
Enter your lucky number (an integer): 33.42
Invalid entry. You’ll be given a random lucky number.
Your lucky number is 8.
```

Figure 15.1 LuckyNumber program, which uses try and catch blocks for numeric user entry

Page 694

In this example, the JVM instantiates an `InputMismatchException` object. The JVM then passes the `InputMismatchException` object to the `catch` block heading’s `e` parameter. Because `e` is declared to be an `InputMismatchException` and `InputMismatchException` is not part of the core Java language, at the top of the program, we need to include:

```
import java.util.InputMismatchException;
```

After passing the exception object to the `catch` block, the JVM executes the `catch` block's body. In this example, the `catch` block prints an "Invalid entry . . ." message and assigns a random number to the `num` variable. Then execution continues with the code below the `catch` block.

Throwing an Exception

When the JVM instantiates an exception object, we say that the JVM *throws an exception*. We'd prefer to say "throws an exception object" rather than "throws an exception" because the thing that's being thrown is an exception object. But most programmers don't worry about the difference between an exception, which is an event, and an exception object. No big deal. We'll go with the flow and use the standard terminology—throwing an exception.

When the JVM throws an exception, the JVM looks for a matching `catch` block. If it finds a matching `catch` block, it executes it. If it does not find a matching `catch` block, the JVM prints the exception object's exception message and terminates the program. What is a "matching `catch` block"? A `catch` block is "matching" if the `catch` heading's parameter type is the same as the type of the thrown exception.[2] For example, in the `LuckyNumber` program, the `InputMismatchException` parameter matches the

`InputMismatchException` object thrown by the `nextInt` method call. Therefore, the `InputMismatchException` parameter's `catch` block is a matching `catch` block if and when the `nextInt` method call throws an `InputMismatchException`.

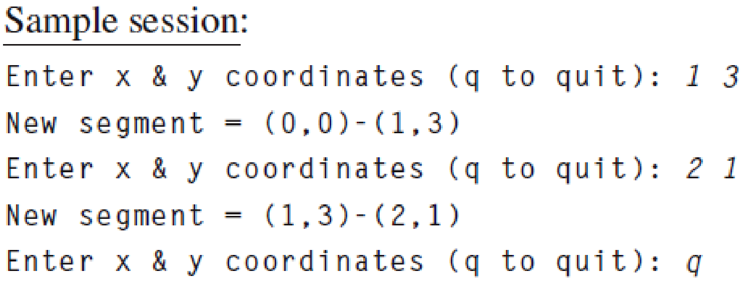
An exception object contains information about the error, including the error's type and a list of the method calls that led to the error. We'll use some of the exception object's information later, but for now, all we need the exception object for is its ability to match up with the proper `catch` block.

15.4 Line Plot Example

Now let's see how `try` and `catch` are used in the context of a more complicated program. We start by presenting a program without `try` and `catch` blocks. Then we analyze the program and determine how it can be improved by adding `try` and `catch` blocks.

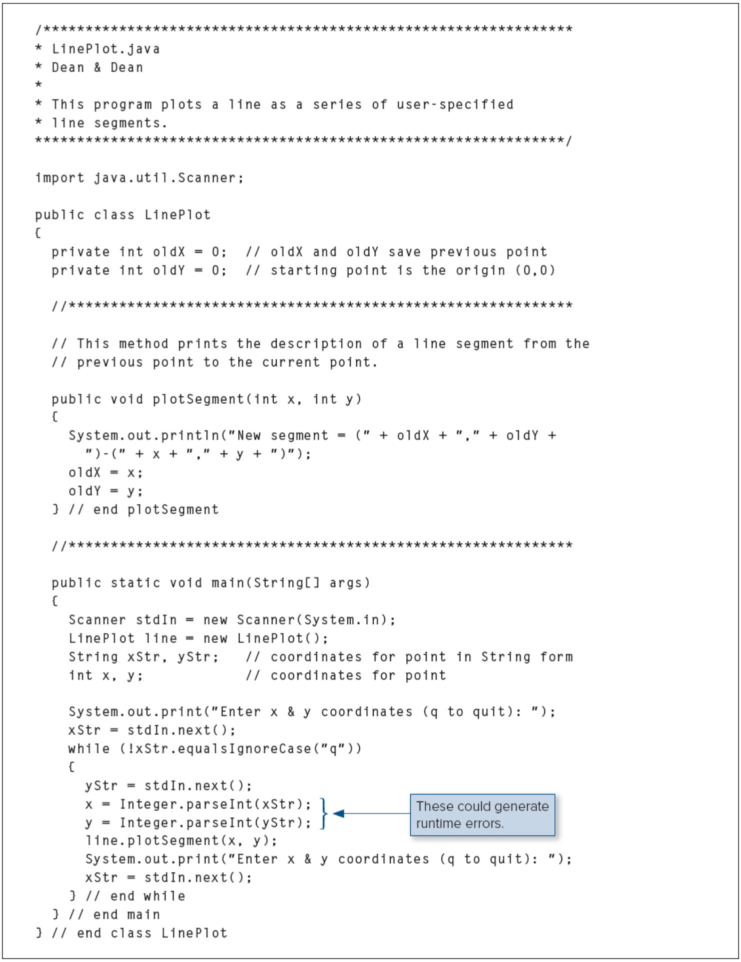
First-Cut LinePlot Program

The program in Figure 15.2 plots a line by reading in coordinate positions for a series of points. The best way to get a handle on what the LinePlot program does is to show a sample session. Below, the user chooses to plot a line that goes from the origin (the default starting point) to point (1,3) to point (2,1):



Sample session:

```
Enter x & y coordinates (q to quit): 1 3
New segment = (0,0)-(1,3)
Enter x & y coordinates (q to quit): 2 1
New segment = (1,3)-(2,1)
Enter x & y coordinates (q to quit): q
```



```
/****************************************************************
* LinePlot.java
* Dean & Dean
*
* This program plots a line as a series of user-specified
* line segments.
***************************************************************/

import java.util.Scanner;

public class LinePlot
{
  private int oldX = 0;  // oldX and oldY save previous point
  private int oldY = 0;  // starting point is the origin (0,0)

  //*************************************************************

  // This method prints the description of a line segment from the
  // previous point to the current point.

  public void plotSegment(int x, int y)
  {
    System.out.println("New segment = (" + oldX + "," + oldY +
      ")-(" + x + "," + y + ")");
    oldX = x;
    oldY = y;
  } // end plotSegment

  //*************************************************************

  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    LinePlot line = new LinePlot();
    String xStr, yStr;   // coordinates for point in String form
    int x, y;            // coordinates for point

    System.out.print("Enter x & y coordinates (q to quit): ");
    xStr = stdIn.next();
    while (!xStr.equalsIgnoreCase("q"))
    {
      yStr = stdIn.next();
      x = Integer.parseInt(xStr);
      y = Integer.parseInt(yStr);
      line.plotSegment(x, y);
      System.out.print("Enter x & y coordinates (q to quit): ");
      xStr = stdIn.next();
    } // end while
  } // end main
} // end class LinePlot
```

These could generate runtime errors.

Figure 15.2 LinePlot program that plots a line—first draft

As you can see, the program's display is very primitive—it uses text to represent each line segment. In a real line-plotting program, you'd use

JavaFX's `LineChart` class to display the line. That's what we do in the GUI section at the end of this chapter. But for now, we'll keep it simple and use a text-based display rather than a GUI-based display. That way, we can maintain focus on this chapter's primary topic, exception handling.

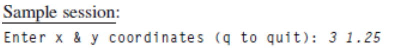
Using "q" as a Sentinel Value

In the past, when you entered numbers inside a loop, you often terminated the loop with a numeric sentinel value. This program employs a more elegant solution because it allows a nonnumeric "q" as the sentinel value. How can you read in numbers and the string "q" with the same input statement? Use strings for both types of input—for the "q" and also for the numbers. For each number input, the program converts the number string to a number by calling the `Integer` class's `parseInt` method.

We described the `Integer` class's `parseInt` method back in Chapter 5. The `parseInt` method attempts to convert a given string to an integer. That should sound familiar; in the LuckyNumber program, we used the `Scanner` class's `nextInt` method to convert a given string to an integer. The difference is that the nextInt method gets its string from a user and the `parseInt` method gets its string from a passed-in parameter. If the passed-in parameter is not the string representation of an integer (with digits and an optional minus sign), the JVM throws a `NumberFormatException`. `NumberFormatException` is in the `java.lang` package. Because the JVM automatically imports the `java.lang` package, your program doesn't need an explicit import to refer to a `NumberFormatException`.

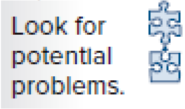
Input Validation

Note how the LinePlot program calls `stdIn.next` to read *x* coordinate and *y* coordinate values into `xStr` and `yStr`, respectively. Then the program attempts to convert `xStr` and `yStr` to integers by calling `Integer.parseInt`. The conversions work fine as long as `xStr` and `yStr` are string representations of integers. But what happens if the user enters a noninteger for xStr or `yStr`? With invalid input, the program crashes, like this:



Sample session:

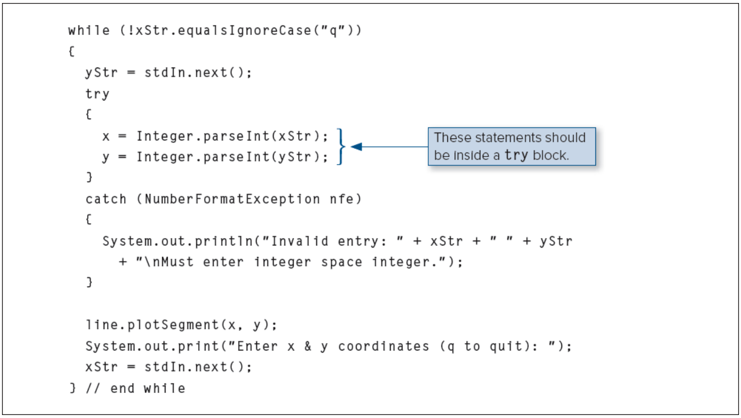
```
Enter x & y coordinates (q to quit): 3 1.25
Exception in thread "main" java.lang.NumberFormatException: For input string: "1.25"
```



Look for potential problems.

To deal with this possibility, let's rewrite the `while` loop in the `main` method of Figure 15.2 so that it includes input validation using a `try-catch` mechanism. The first step is to identify the dangerous code. Can you find the dangerous code? The two parseInt method calls are dangerous in that they might throw a NumberFormatException. So let's put those two statements into a try block and add a matching catch block, as shown in Figure 15.3.

Do you see any logic errors in Figure 15.3's `while` loop? What happens if there's invalid input? A `NumberFormatException` object is thrown and caught, and then an error message is printed. Then `line.plotSegment` executes. But you wouldn't want to print the line segment if the input values were messed up. To avoid that possibility, move the line.plotSegment(x, y); line to the last line in the `try` block. This way, it gets executed only if the two `parseInt` method calls work properly. Figure 15.4 shows the final version of the LinePlot program's `while` loop.

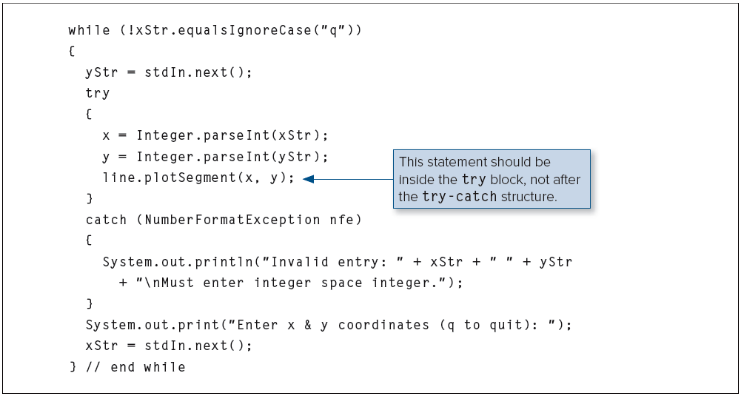


```
while (!xStr.equalsIgnoreCase("q"))
{
  yStr = stdIn.next();
  try
  {
    x = Integer.parseInt(xStr);
    y = Integer.parseInt(yStr);
  }
  catch (NumberFormatException nfe)
  {
    System.out.println("Invalid entry: " + xStr + " " + yStr
      + "\nMust enter integer space integer.");
  }

  line.plotSegment(x, y);
  System.out.print("Enter x & y coordinates (q to quit): ");
  xStr = stdIn.next();
} // end while
```

These statements should be inside a `try` block.

Figure 15.3 First attempt at improving the LinePlot program's while loop

Page 698



```
while (!xStr.equalsIgnoreCase("q"))
{
  yStr = stdIn.next();
  try
  {
    x = Integer.parseInt(xStr);
    y = Integer.parseInt(yStr);
    line.plotSegment(x, y);
  }
  catch (NumberFormatException nfe)
  {
    System.out.println("Invalid entry: " + xStr + " " + yStr
      + "\nMust enter integer space integer.");
  }
  System.out.print("Enter x & y coordinates (q to quit): ");
  xStr = stdIn.next();
} // end while
```

This statement should be inside the `try` block, not after the `try-catch` structure.

Figure 15.4 Final version of the LinePlot program's while loop

15.5 try Block Details

Now that you know the basic idea

behind `try` blocks, it's time to flesh out some subtle `try` block details.

try Block Size

Deciding on the size of your `try` blocks is a bit of an art. Sometimes it's better to use small `try` blocks, and sometimes it's better to use larger `try` blocks. It's legal to surround an entire method body with a try block, but that's usually counterproductive because then dangerous code is harder to identify. In general, you should make your `try` blocks small enough so that your dangerous code is easily identified.

On the other hand, if you need to execute several related dangerous statements in succession, you should consider surrounding the statements with one inclusive `try` block rather than surrounding each statement with its own small `try` block. Multiple small `try` blocks can lead to cluttered code. One inclusive `try` block can lead to improved readability. The improved LinePlot program includes both `parseInt` statements in a single `try` block because they are conceptually related and physically close together. That improves readability.

Assume That try Block Statements Are Skipped

If an exception is thrown, the JVM immediately jumps out of the current `try` block. The immediacy of the jump means that if there are statements in the `try` block after the exception-throwing statement, those statements get skipped. The compiler is a pessimist. It knows that statements inside a `try` block might possibly be skipped, and it assumes the worst; that is, it assumes that all statements inside a `try` block get skipped.

Consequently, if there's a `try` block that contains an assignment to *x*, the compiler assumes that the assignment is skipped. If there's no assignment to x outside of the `try` block and *x*'s value is needed outside of the `try` block, you'll get this compile-time error:

variable x might not have been initialized ⚠

If you get that error, usually you can fix it by initializing the variable prior to the `try` block. Let's look at an example. . . .

Your goal is to implement a `getIntFromUser` method that performs robust input for an `int` value. Your method should prompt the user for an integer, read the entered value as a string, and then convert the string to an `int`. If the conversion fails, your method should reprompt the user for an integer. If the user eventually enters a valid integer value, `getIntFromUser` should return it to the calling module.

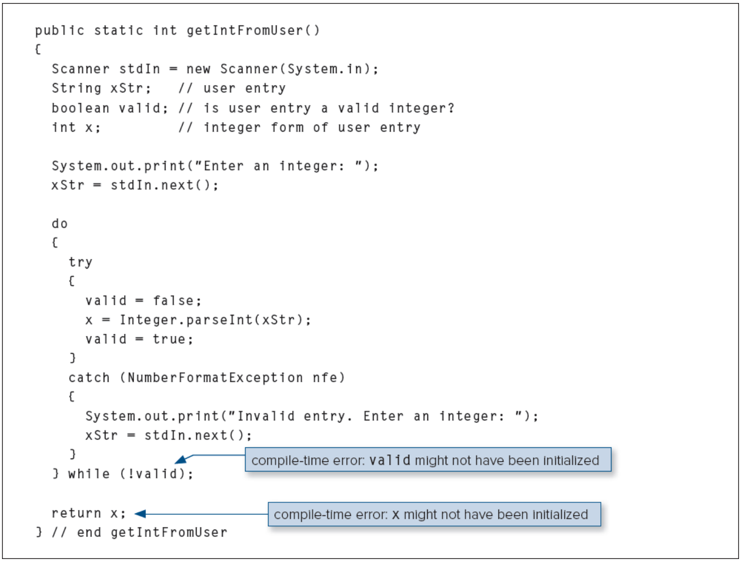
Figure 15.5 is a first-cut attempt at implementing the `getIntFromUser` method. It does a good job with the logic, but it contains compile-time errors that are due to the initializations inside the `try` block. We'll fix the `try` block's errors soon enough, but let's first explain The `try` block's logic.

the `try` block contains these three lines:

```
valid = false;
x = Integer.parseInt(xStr);
valid = true;
```

Assume one thing, then change as required.

Note how the three-line code fragment assigns `valid` to `false` and then turns around and assigns it back to `true`. Strange, eh? Actually, it's a fairly common strategy to assume one thing, try it out, and then change the assumption if it’s proven wrong. And that’s what’s happening here. This code starts by assuming that the user entry is invalid. It calls parseInt to test whether it’s actually valid; that is, it checks to see if the user entry is an integer. If it is valid, the next statement executes, and valid gets set to true. But what happens if the parseInt conversion fails? The valid variable never gets set to `true` because an exception is thrown and the JVM immediately jumps out of the `try` block. So this code seems reasonable. Unfortunately, “seems reasonable” isn’t good enough this time.



```
public static int getIntFromUser()
{
  Scanner stdIn = new Scanner(System.in);
  String xStr;   // user entry
  boolean valid; // is user entry a valid integer?
  int x;         // integer form of user entry

  System.out.print("Enter an integer: ");
  xStr = stdIn.next();

  do
  {
    try
    {
      valid = false;
      x = Integer.parseInt(xStr);
      valid = true;
    }
    catch (NumberFormatException nfe)
    {
      System.out.print("Invalid entry. Enter an integer: ");
      xStr = stdIn.next();
    }
  } while (!valid);

  return x;
} // end getIntFromUser
```

compile-time error: `valid` might not have been initialized

compile-time error: `x` might not have been initialized

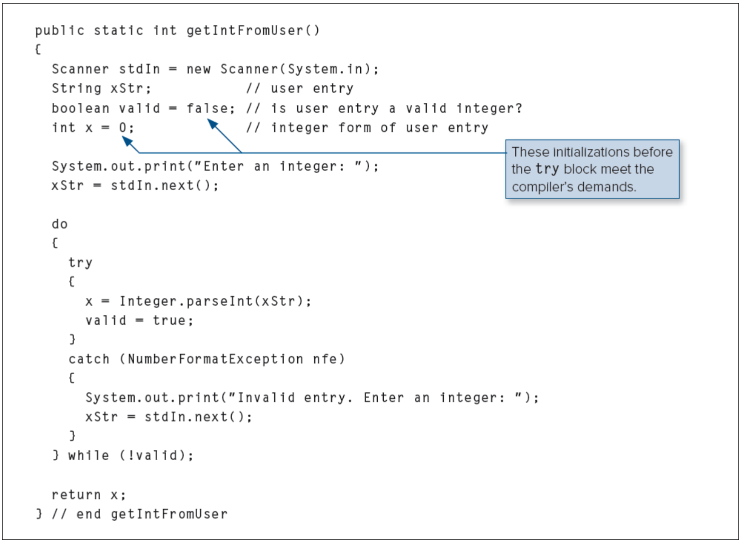
Figure 15.5 A method that illustrates the problem with initializing inside a try block

Can you figure out the compile-time errors? If not, don’t feel bad; we didn’t see them until after the compiler helped us. As shown by the callouts in Figure 15.5, the compiler complains that the `valid` and `x` variables might not have been initialized. Why all the fuss? Can’t the compiler see that `valid` and `x` are assigned values in the `try` block? Yes, the compiler can see the assignments, but remember that the compiler is a pessimist. It assumes that all statements inside a `try` block are skipped. Even though we know that the `valid = false;` statement is in no actual danger of being skipped (it's a simple assignment, and it's the first line in the `try` block), the compiler still assumes that it gets skipped.

What's the solution? (1) Move the valid = false; assignment up to `valid`'s declaration line. (2) Initialize x to 0 as part of x's declaration line. Figure 15.6 contains the corrected implementation.

15.6 Two Categories of Exceptions—Checked and Unchecked

Exceptions fall into two categories—*checked* and *unchecked.* Checked exceptions must be checked with a `try-catch` mechanism. Unchecked exceptions can optionally be checked with a try-catch mechanism, but it's not a requirement.



```
public static int getIntFromUser()
{
  Scanner stdIn = new Scanner(System.in);
  String xStr;           // user entry
  boolean valid = false; // is user entry a valid integer?
  int x = 0;             // integer form of user entry

  System.out.print("Enter an integer: ");
  xStr = stdIn.next();

  do
  {
    try
    {
      x = Integer.parseInt(xStr);
      valid = true;
    }
    catch (NumberFormatException nfe)
    {
      System.out.print("Invalid entry. Enter an integer: ");
      xStr = stdIn.next();
    }
  } while (!valid);

  return x;
} // end getIntFromUser
```

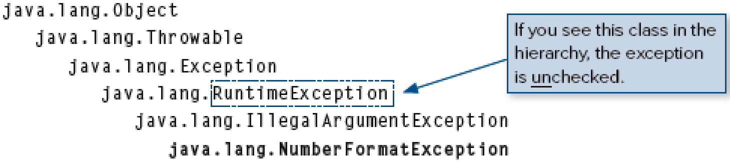
These initializations before the `try` block meet the compiler's demands.

Figure 15.6 Corrected version of the getIntFromUser method in Figure 15.5

Identifying an Exception's Category

How can you tell whether a particular exception is classified as checked or unchecked? An exception is an object, and as such, it is associated with a particular class. To find out if a particular exception is checked or unchecked, look up its associated class on Oracle’s Java API website. Once you find the class, look at its ancestors. If you find that it’s a descendant of the `RuntimeExeption` class, then it’s an unchecked exception. Otherwise, it’s a checked exception.

For example, if you look up `NumberFormatException` on Oracle’s Java API website, you’ll see this:



```
java.lang.Object
   java.lang.Throwable
      java.lang.Exception
         java.lang.RuntimeException
            java.lang.IllegalArgumentException
               java.lang.NumberFormatException
```

If you see this class in the hierarchy, the exception is unchecked.

This shows that the `NumberFormatException` class is a descendant of the `RuntimeException` class, so the `NumberFormatException` class is an unchecked exception.

Figure 15.7 shows the class

hierarchy for all exceptions. It reiterates the point that unchecked exceptions are descendants of the `RuntimeException` class. It also shows that some unchecked exceptions are descendants of the `Error` class. In the interest of keeping things simple, we didn't mention the `Error` class previously. You probably won't encounter its exceptions unless you do a lot of recursive programming, as described in Chapter 11. In that case you probably will encounter `StackOverflowException`, which derives from `VirtualMachineError`, which derives from `Error`.

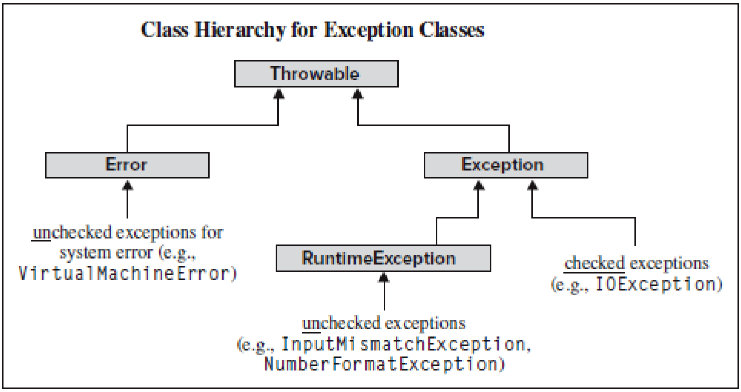

Class Hierarchy for Exception Classes
Throwable
Error
Exception
unchecked exceptions for system error (e.g., VirtualMachineError)
RuntimeException
checked exceptions (e.g., IOException)
unchecked exceptions (e.g., InputMismatchException, NumberFormatException)

Figure 15.7 Exception class hierarchy

Page 702

Programmer-Defined Exception Classes

It’s possible for programmers to define their own exception classes. Such programmer-defined exception classes must be derived from the `Exception` class or from a subclass of the `Exception` class. Generally speaking, you should limit yourself to predefined exception classes, because programmer-defined exception classes tend to fragment error-handling activities, and that makes programs harder to understand.

15.7 Unchecked Exceptions

As you learned in the previous section,

unchecked exceptions need not be checked with a `try-catch` mechanism. However, at runtime, if the JVM throws an unchecked exception and there's no `catch` block to catch it, the program will crash.

Strategies for Handling Unchecked Exceptions

If your program contains code that might throw an unchecked exception, there are two alternate strategies for dealing with it:

1. Use a `try-catch` structure.

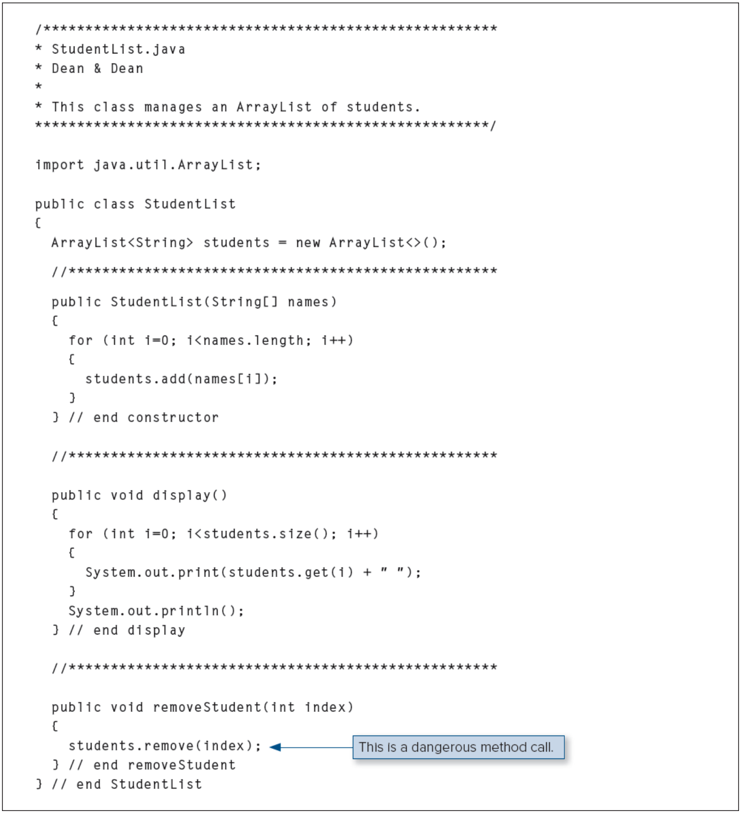
or

2. Don't attempt to catch the exception, but write the code carefully so as to avoid the possibility of the exception being thrown.

In the `getIntFromUser` method in Figure 15.6, we employed the first strategy—we used a `try-catch` structure to handle the dangerous `parseInt` method call. Normally, you should use a try-catch structure for parse method calls (`parseInt`, `parseLong`, `parseDouble`, and so on) because that leads to cleaner solutions. In the next example, the preferred strategy isn't so clear cut. We'll use both strategies and compare the results.

StudentList Example

Figure 15.8 presents a `StudentList` class that manages a list of student names. The class stores student names in an `ArrayList` named `students`. The class contains a constructor for initializing the `students` list, a `display` method for printing the `students` list, and a `removeStudent` method that removes a specified student from the `students` list. We'll focus on the `removeStudent` method.

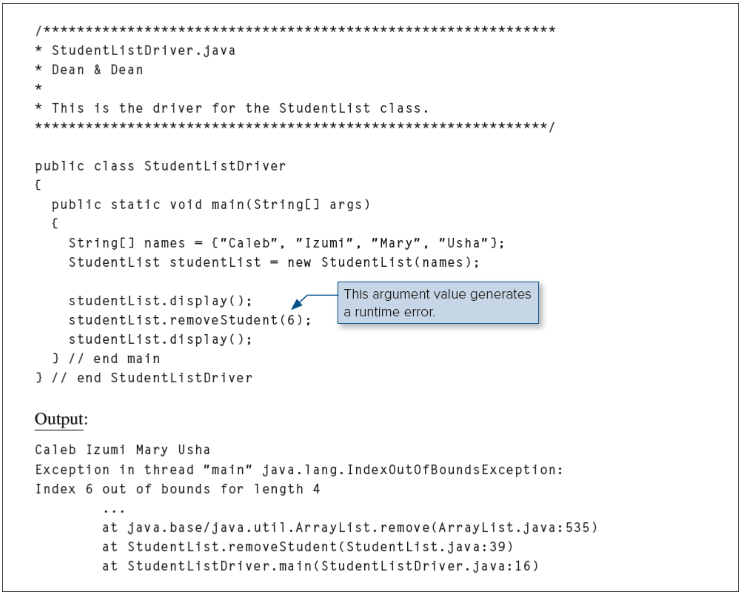


```
/******************************************************
* StudentList.java
* Dean & Dean
*
* This class manages an ArrayList of students.
******************************************************/

import java.util.ArrayList;

public class StudentList
{
  ArrayList<String> students = new ArrayList<>();

  //***************************************************

  public StudentList(String[] names)
  {
    for (int i=0; i<names.length; i++)
    {
      students.add(names[i]);
    }
  } // end constructor

  //***************************************************

  public void display()
  {
    for (int i=0; i<students.size(); i++)
    {
      System.out.print(students.get(i) + " ");
    }
    System.out.println();
  } // end display

  //***************************************************

  public void removeStudent(int index)
  {
    students.remove(index);
  } // end removeStudent
} // end StudentList
```

This is a dangerous method call.

Figure 15.8 First draft of StudentList class, which maintains a list of students

The `students.remove` method call is dangerous because it might throw an unchecked exception, ⚠ `IndexOutOfBoundsException`. If its `index` argument holds the index of one of the `students`' elements, then that element is removed from the students ArrayList. But if its `index` argument holds an invalid index, an `IndexOutOfBoundsException` is thrown. This occurs, for example, if we use Figure 15.9's StudentListDriver class as the driver. Note how the `StudentListDriver` class uses an index value of 6, even though there are only four students in the student list. The `StudentListDriver` and `StudentList` classes compile just fine, but when run, the `students.remove` method call throws an exception and the JVM terminates the program and prints the error message shown at the bottom of Figure 15.9.



```
/*******************************************************************
* StudentListDriver.java
* Dean & Dean
*
* This is the driver for the StudentList class.
*******************************************************************/

public class StudentListDriver
{
  public static void main(String[] args)
  {
    String[] names = {"Caleb", "Izumi", "Mary", "Usha"};
    StudentList studentList = new StudentList(names);

    studentList.display();
    studentList.removeStudent(6);
    studentList.display();
  } // end main
} // end StudentListDriver
```

This argument value generates a runtime error.

Output:

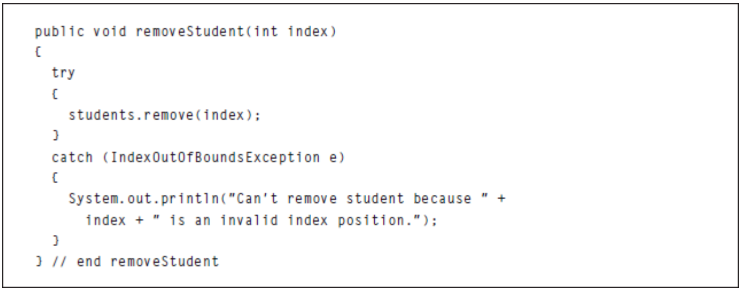
```
Caleb Izumi Mary Usha
Exception in thread "main" java.lang.IndexOutOfBoundsException:
Index 6 out of bounds for length 4
        ...
        at java.base/java.util.ArrayList.remove(ArrayList.java:535)
        at StudentList.removeStudent(StudentList.java:39)
        at StudentListDriver.main(StudentListDriver.java:16)
```

Figure 15.9 Driver of StudentList class

Improve the removeStudent Method

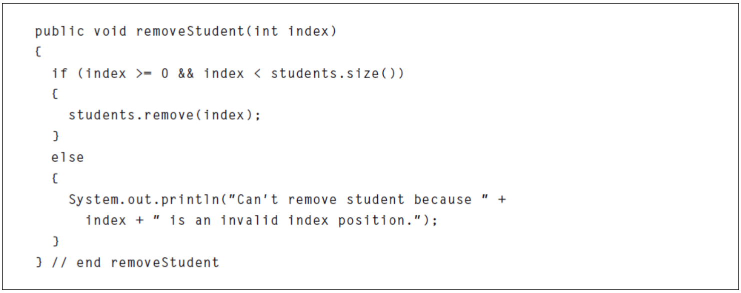
Make programs robust.

Let's now make the removeStudent method more robust by gracefully handling the case where it's called with an invalid index. Figures 15.10a and 15.10b show two different robust implementations for the removeStudent method. The first implementation uses a try-catch mechanism and the second implementation uses careful code. These are the two strategies mentioned earlier for handling unchecked exceptions.



```
public void removeStudent(int index)
{
  try
  {
    students.remove(index);
  }
  catch (IndexOutOfBoundsException e)
  {
    System.out.println(”Can’t remove student because ” +
      index + ” is an invalid index position.”);
  }
} // end removeStudent
```

Figure 15.10a Using a try-catch structure for the removeStudent method



```
public void removeStudent(int index)
{
  if (index >= 0 && index < students.size())
  {
    students.remove(index);
  }
  else
  {
    System.out.println(”Can’t remove student because ” +
      index + ” is an invalid index position.”);
  }
} // end removeStudent
```

Figure 15.10b Using a careful-code strategy for the removeStudent method

Which solution is better—a `try-catch` mechanism or careful code?

The solutions are about the same in terms of readability. With things being equal in terms of readability, go with the careful-code implementation because it's more efficient. Exception handling code is less efficient because it requires the JVM to instantiate an exception object and find a matching catch block.

15.8 Checked Exceptions

Let's now look at checked exceptions. If a code fragment has the potential of throwing a checked exception, the compiler forces you to associate that code fragment with a `try-catch` mechanism. If there is no associated `try-catch` mechanism, the compiler generates an error. With unchecked exceptions, you have a choice of how to handle them—a `try-catch` mechanism or careful code. With checked exceptions, there's no choice

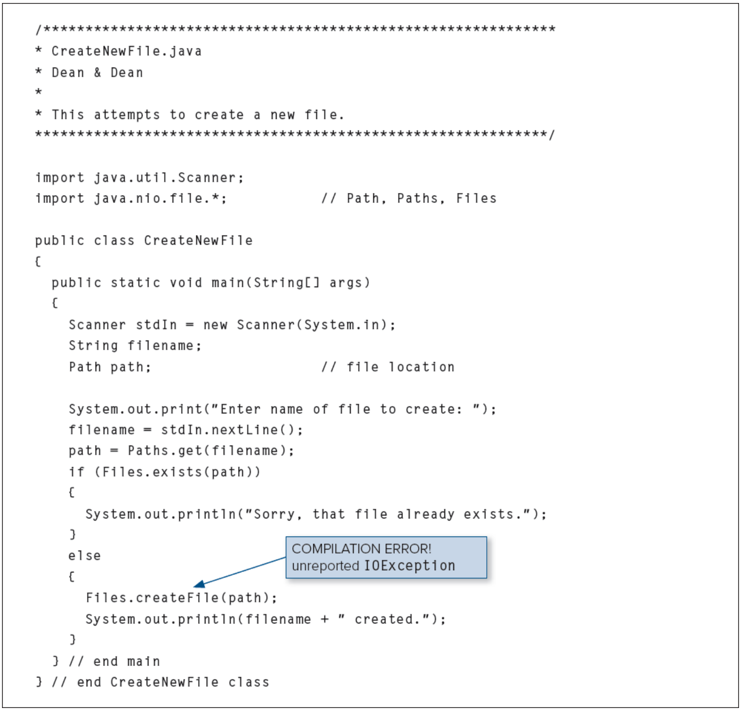
—you must use a `try-catch` mechanism.

CreateNewFile Program

Page 706

For good examples of checked exceptions, we need files. The next chapter explains file details, but you don't need to know file details now. You can understand how to deal with file-code exceptions before you understand file-code details. The draft of a CreateNewFile program in Figure 15.11 attempts to create an empty file with a user-specified name. To do this, the program needs to use the `Path`, `Paths` and `Files` classes in the API's `java.nio.file` package. So it imports that package.

This program is supposed to prompt the user for the name of a file to create. If a file with that name already exists, it is supposed to print "Sorry, that file already exists." If a file with that name does not already exist, it is supposed to create a new file with that name. Unfortunately, when we try to compile the code in Figure 15.11, the complier complains:



```
/*************************************************************
* CreateNewFile.java
* Dean & Dean
*
* This attempts to create a new file.
*************************************************************/

import java.util.Scanner;
import java.nio.file.*;           // Path, Paths, Files

public class CreateNewFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String filename;
    Path path;                    // file location

    System.out.print("Enter name of file to create: ");
    filename = stdIn.nextLine();
    path = Paths.get(filename);
    if (Files.exists(path))
    {
      System.out.println("Sorry, that file already exists.");
    }
    else
    {
      Files.createFile(path);
      System.out.println(filename + " created.");
    }
  } // end main
} // end CreateNewFile class
```

COMPILATION ERROR!
unreported `IOException`

Figure 15.11 Draft of CreateNewFile program, which is supposed to create a new file

```
<path-to-source-code-file>: 28: error: unreported exception IOException;
must be caught or declared to be thrown
  Files.createFile(path);
                  ^
```

Page 707

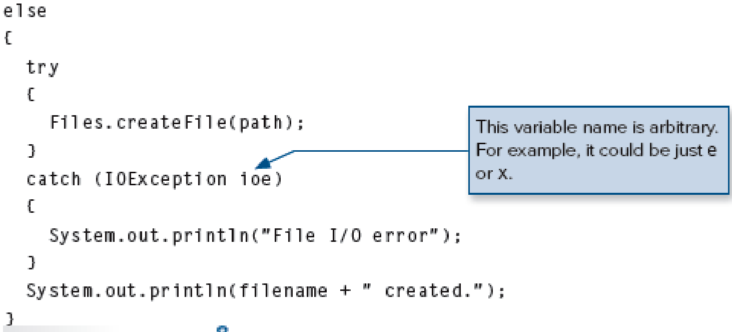
This tells us that the `createFile`

method throws a checked exception, specifically, the `IOException`. The "quick and dirty" way to satisfy the compiler is to append throws IOException to the method header, as suggested back in Chapter 3, in Section 3.24.

A more responsible way to satisfy the compiler is to provide another import:

import java.io.IOException;

Then surround the `createFile` method call with a `try` block like this:



```
else
{
  try
  {
    Files.createFile(path);
  }
  catch (IOException ioe)
  {
    System.out.println("File I/O error");
  }
  System.out.println(filename + " created.");
}
```

This variable name is arbitrary. For example, it could be just e or x.

Don't move on until you are sure of your solution.

This results in a program that compiles successfully and runs. But is it a good program? Novice programmers often solve problems by trying something out without thoroughly thinking it through, and if it leads to reasonable results, they quickly move on. Try to resist that urge.

Although the above code compiles and runs, it doesn't behave appropriately when an `IOException` is thrown. Can you identify the inappropriate behavior? If an `IOException` is thrown, the `catch` block prints the message:

"File I/O error"

But then, even though the file was not created, the program also prints the message:

filename + " created."

Remember: Just because a program runs, that doesn't mean it's correct. Also, just because you've fixed one problem, that doesn't mean you've fixed all problems. Read on . . .

Using API Documentation When Writing Exception Handling Code

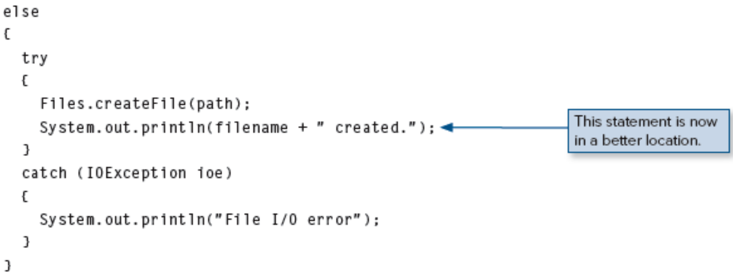
When you want to use a method or constructor from one of the API classes and you're not sure about it, you can look it up in the API documentation. On its API documentation page, look for a "throws" section. This identifies specific types of exceptions that constructor or method might throw. For an explanation of a specific exception, click on that exception's link. This takes you to the API documentation for that exception's class. On the exception class's API page, scroll down and read the class's description. Then scroll back up and look at the class's class hierarchy. As indicated earlier, if `RuntimeException` is an ancestor, the exception is an unchecked exception. Otherwise, it's a checked exception.

If you apply this API-lookup strategy to the CreateNewFile program, you'll learn the following about the exceptions it might throw:

- The `Paths.get(filename)` method call throws an `InvalidPathException` if the `String` argument cannot be converted to a valid `Path`. The `InvalidPathException` class derives from the `RuntimeException` class, so it's an unchecked exception.
- The `Files.exists(path)` method call throws a `SecurityException`. This derives from the `RuntimeException` class, so it's an unchecked exception.
- The `Files.createFile(path)` method throws an `UnsupportedOperationsException`

in addition to a `SecurityException`. Because these both derive from the `RuntimeException` class, they are unchecked exceptions. It also throws a `FileAlreadyExistsException` and an `IOException`. Since these do not derive from the RuntimeException class, they are checked exceptions, and they must be in a `try` block. Since the `FileAlreadyExistsException` derives from the `IOException` class, a `catch` block that catches an `IOException` will catch them both. The `FileAlreadyExistsException` is the checked exception we thought we handled but really didn’t.

The correct way to handle the two checked exceptions is to move the “created” printout up into the `try` block, after the statement that creates the new file, like this:



```
else
{
  try
  {
    Files.createFile(path);
    System.out.println(filename + " created.");
  }
  catch (IOException ioe)
  {
    System.out.println("File I/O error");
  }
}
```

This statement is now in a better location.

Now the program prints the “created” message only if the file is actually created. Yeah!

15.9 Generic catch Block with Exception Class

Previous programs have used the `try-catch` mechanism to catch only one type of exception. Because the `FileAlreadyExistsException` derives from the `IOException`, the `catch` block in the CreateNewFile program is able to catch either of these two exceptions, However, the CreateNewFile program used an `if` statement to avoid the first of these two exceptions. So it, too, used the `try-catch` mechanism to catch only one type of exception.

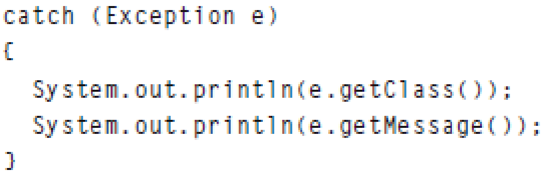
Page 709

Alternatively, we can use exception handling alone to catch more than one exception and distinguish them one from another. There are two ways to do this: (1) Provide one generic `catch` block that catches every type of exception that might be thrown, and in that block provide code that identifies the exception actually thrown. (2) Provide a sequence of `catch` blocks that catch different exceptions. This section describes the generic-`catch`-block technique. The next section describes the sequence-of-`catch`-blocks technique. In practice, as we develop a program, we typically start with the generic-`catch`-block technique and then perhaps move to the sequence-of-`catch`-blocks technique.

Generic catch Block

To provide a generic `catch` block, define a `catch` block with an `Exception` type parameter. Then, inside the `catch` block, call the `getClass` method (inherited by `Exception` from `Object`), and also call the `getMessage` method (inherited by Exception from Throwable). The getClass method call identifies the particular type of exception that was thrown. The getMessage method call explains why that exception was thrown. Just one of these two method calls might leave you wondering, but the combination of both of them usually tells you exactly what you need to know.3

Here is what the generic `catch` block looks like:



```
catch (Exception e)
{
  System.out.println(e.getClass());
  System.out.println(e.getMessage());
}
```

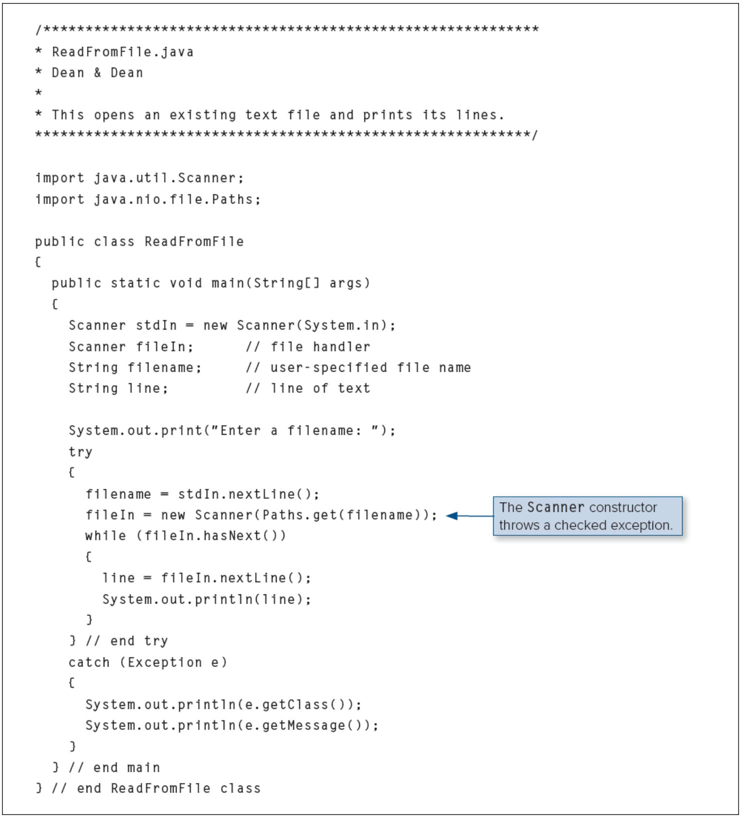
Because the Exception class is in the always available java.lang package, you do not need to import it. If a catch block uses an Exception parameter, it matches all thrown exceptions. Why? Because when an exception is thrown, it looks for a catch parameter that's either identical to the thrown exception or a superclass of the thrown exception. The Exception class is the superclass of all checked exceptions. Therefore, all checked exceptions consider an Exception catch parameter to be a match.

Figure 15.7 showed that the Exception class also includes unchecked exceptions. So in addition to catching and describing all checked exceptions, except for rare Error exceptions, the generic `catch` block above catches and describes all unchecked exceptions thrown from the corresponding try block.

ReadFromFile Example

Figure 15.12a's ReadFromFile program opens a user-specified file and prints all lines of text in that file. The `Scanner` constructor that instantiates the stdIn object does not throw any exceptions. The `System.out.print` method does not throw any exceptions. So we do not include them in the `try` block.

The `stdIn.nextLine` method call throws an unchecked `NoSuchElementException` and an unchecked `IllegalStateException`. As indicated earlier, the `Paths.get(filename)` method call throws an unchecked `InvalidPathException`. Because it takes extra code to check for these problems, and our generic `catch` clearly identifies them if they occur, we (optionally) include these method calls in our enlarged `try` block. The Scanner constructor that instantiates the `fileIn` object throws a checked `IOException`, so we (necessarily) include it in the `try` block.



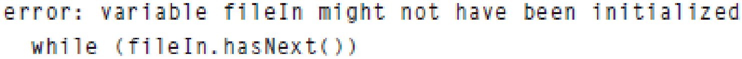
```
/*************************************************************
* ReadFromFile.java
* Dean & Dean
*
* This opens an existing text file and prints its lines.
*************************************************************/

import java.util.Scanner;
import java.nio.file.Paths;

public class ReadFromFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Scanner fileIn;      // file handler
    String filename;     // user-specified file name
    String line;         // line of text

    System.out.print(”Enter a filename: ”);
    try
    {
      filename = stdIn.nextLine();
      fileIn = new Scanner(Paths.get(filename));
      while (fileIn.hasNext())
      {
        line = fileIn.nextLine();
        System.out.println(line);
      }
    } // end try
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end ReadFromFile class
```

The Scanner constructor throws a checked exception.

Figure 15.12a ReadFromFile program—a simple file reader

The `while` loop iterates through all

lines of text in the file. The `fileIn.hasNext` method throws an unchecked `IllegalStateException`, and the `fileIn.nextLine` method throws an unchecked IllegalStateException and an unchecked `NoSuchElementException`. Because these exceptions are unchecked, one might think we could remove the code that throws them from the `try` block—perhaps putting the `while` loop after the `catch` block. But if we attempt that, the compiler complains:

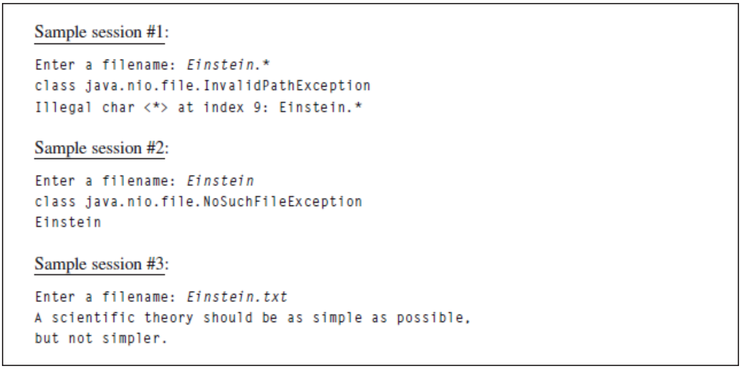


```
error: variable fileIn might not have been initialized
  while (fileIn.hasNext())
```

Page 711

This is another example of the principle we discussed back in Section 15.5. The compiler is a pessimist. It knows that statements inside a `try` block might be skipped, and it assumes the worst. In the Section 15.5 example, we were able to move the required initialization out of the `try` block and put it before the `try` block. But this time, we can't do that, because the initialization throws a checked exception. Therefore, we must include the `while` loop in the try block along with that initialization.

Figure 15.12b contains outputs from three separate executions of the ReadFromFile program in Figure 15.12a. In the first sample session, the user employs an illegal character in the filename specification. The `e.getClass` method call says this input generates an `InvalidPathException`. Java API documentation and the first bullet in the last subsection of Section 15.8 say this exception must have come from the `Paths.get` method call.



Sample session #1:

```
Enter a filename: Einstein.*
class java.nio.file.InvalidPathException
Illegal char <*> at index 9: Einstein.*
```

Sample session #2:

```
Enter a filename: Einstein
class java.nio.file.NoSuchFileException
Einstein
```

Sample session #3:

```
Enter a filename: Einstein.txt
A scientific theory should be as simple as possible,
but not simpler.
```

Figure 15.12b Outputs from the ReadFromFile program in Figure 15.12a

In the second sample session, the user specifies a nonexistent file. The `e.getClass` method says that this input generates a `NoSuchFileException`. This `Class` information is right “on target” and quite helpful, but it’s not one of the exceptions that API documentation associates with our program’s constructor and method calls. What’s happening here? `NoSuchFileException` derives from the `IOException` that API documentation says the `Scanner(Path source)` constructor throws. So the actual output is more specific than the API documentation. The Java compiler is getting smarter as it gets older!

In the third sample session, the user correctly specifies the file, and the program correctly displays its contents.

15.10 Multiple `catch` Blocks and Multiple Exceptions per Block

When a `try` block might throw more than one type of exception, instead of providing one generic `catch` block, you can provide a sequence of `catch` blocks. Multiple `catch` blocks help you provide different responses for different types of exceptions. A sequence of `catch` blocks is like the "if, else if, else if, . . ." form of the `if` statement. In any particular situation, the first `catch` with a matching parameter is the one that executes. If the final `catch` block is generic (and can catch any type of exception), it is like the "else" part of an "if, else if, . . . else" statement.

When you are developing a new program, it's convenient to start with a generic `catch` block and insert specific `catch` blacks ahead of it as you discover the need to respond differently to different kinds of exceptions.

The ReadFromFile Program Revisited

The ReadFromFile program in Figure 15.12a showed how you can use the compiler's knowledge and experiment with a simple generic catch block to discover the types of exceptions that might be thrown by various forms of bad input. Once you have discovered all the types of exceptions bad user entries might generate, you can put these particular exceptions into an additional `catch` block inserted before the generic catch block. Then, you can modify the code so that if any of the bad entries occurs, the program prints a short error message, repeats the original prompt, and takes new input—in an iteration that continues until the input is satisfactory. The ReadFromFile2 program in Figure 15.13a includes these modifications.

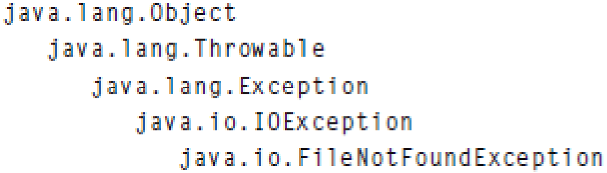
In the ReadFromFile2 program, we saved four lines of code by eliminating the `filename` and `line` variables. Then we added the variable, `boolean makeEntry`, which is initially `true`. This new variable allows entry into the outer `while` loop, which includes all other code. When the file opens successfully, execution proceeds through to the `makeEntry = false;` statement. This allows the current iteration to continue, but it prevents another iteration.

Notice how the new `catch` block's heading includes two exceptions types. The single "or" symbol, |, which separates them says either of them will be caught by the same `catch` block. Thus, this one block catches all plausible user-entry errors and prints a message that applies to any of them. Assuming user entries like those in Figure 15.12b, Figure 15.13b shows the output generated in just one execution of the ReadFromFile2 program in Figure 15.13a.

catch Block Ordering—The Order Matters

Whenever you use more than one `catch` block, and the class of one `catch` block's exception is derived from the class of another `catch` block's exception, you must arrange the `catch` blocks so that the more general exceptions come later.

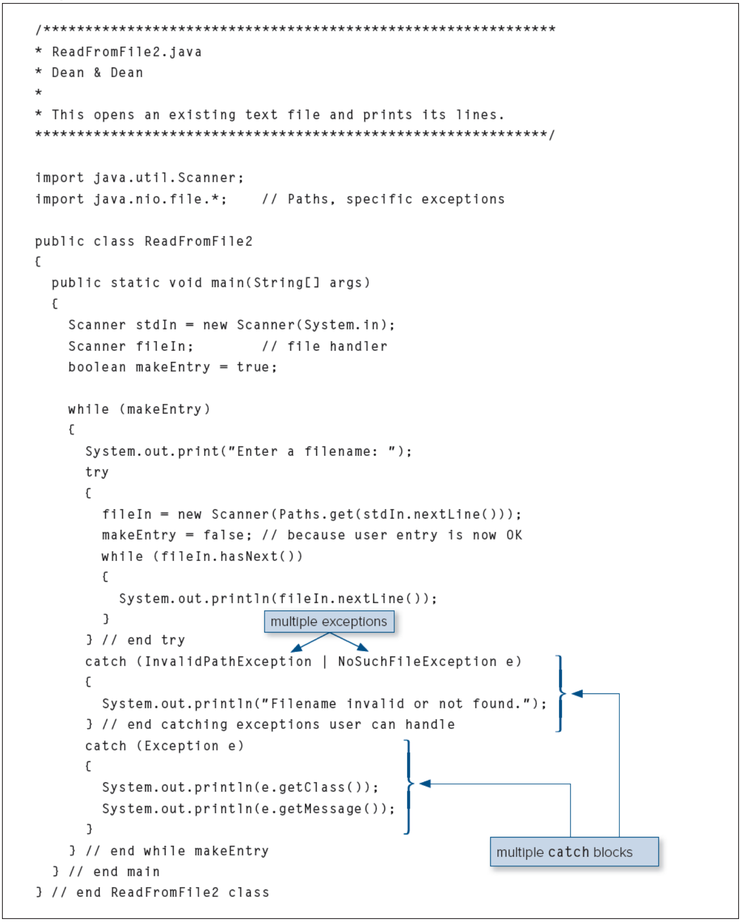
For example, if you look up `FileNotFoundException` on the Java API website, you'll see this hierarchy:



```
java.lang.Object
   java.lang.Throwable
      java.lang.Exception
         java.io.IOException
            java.io.FileNotFoundException
```

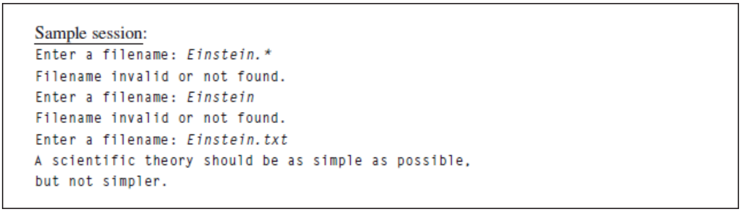
⚠ If you choose to have a `catch` block with a `FileNotFoundException` and a `catch` block with an `IOException` in the same `catch`-block sequence, you must put the `IOException` block after the FileNotFoundException block. If you put a catch block for IOException first, it would match FileNotFoundException too, and the other catch block would always be skipped. As long as you understand these principles, there's no need to memorize the hierarchical relationships among all types of exceptions because the compiler will tell you in a compile-time error if you try to arrange multiple catch blocks in the wrong order.

Page 713



```
/*************************************************************
* ReadFromFile2.java
* Dean & Dean
*
* This opens an existing text file and prints its lines.
*************************************************************/

import java.util.Scanner;
import java.nio.file.*;    // Paths, specific exceptions

public class ReadFromFile2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Scanner fileIn;         // file handler
    boolean makeEntry = true;

    while (makeEntry)
    {
      System.out.print(”Enter a filename: ”);
      try
      {
        fileIn = new Scanner(Paths.get(stdIn.nextLine()));
        makeEntry = false; // because user entry is now OK
        while (fileIn.hasNext())
        {
          System.out.println(fileIn.nextLine());
        }
      } // end try
      catch (InvalidPathException | NoSuchFileException e)
      {
        System.out.println(”Filename invalid or not found.”);
      } // end catching exceptions user can handle
      catch (Exception e)
      {
        System.out.println(e.getClass());
        System.out.println(e.getMessage());
      }
    } // end while makeEntry
  } // end main
} // end ReadFromFile2 class
```

Figure 15.13a ReadFromFile2 program—an improved file reader



Sample session:

```
Enter a filename: Einstein.*
Filename invalid or not found.
Enter a filename: Einstein
Filename invalid or not found.
Enter a filename: Einstein.txt
A scientific theory should be as simple as possible,
but not simpler.
```

Figure 15.13b Output produced by the ReadFromFile2 program in Figure 15.13a

The compiler does not allow you to put an exception and an ancestor of that exception in the same catch block. For example, it will complain if you try to put a FileNotFoundException and an IOException together in the same multiple-exception catch header. However, the compiler did not complain when we put an InvalidPathException and a NoSuchFileException together in the same catch-block parameter list in Figure 15.13a. So what can you infer about the relationship between these last two exceptions? Since the compiler accepts them both in the same catch header, neither of these two classes is a subclass of the other.

Removing Redundant Generic catch and Handling Unchecked Exceptions

Earlier, we suggested starting program development with a generic catch block and inserting specific catch blocks ahead of it as the compiler identifies exceptions that need specific responses. As development proceeds, newly introduced specific catch blocks will catch more and more checked exceptions. If specific catch blocks catch all checked exceptions, the generic catch block is redundant. If the generic catch block becomes redundant, you should simplify the program by removing it.

The above procedure deals with checked exceptions, but it does not deal with unchecked exceptions. Any time after your program compiles, you can start looking for unchecked exceptions by executing it with different types of input. If your program still has a generic catch block, that catch block will catch an unchecked RuntimeException as well as all checked exceptions. If you have removed the generic catch block, the JVM will still identify any unchecked exception that might occur. In either event, you should try to deal with possible unchecked exceptions by using "careful code"; that is, rewriting your code so that situations which might generate unchecked exceptions never occur. Code that deals with possible unchecked exceptions may be in a try block, but it should not involve the relatively inefficient catch mechanism.

15.11 Understanding Exception Messages

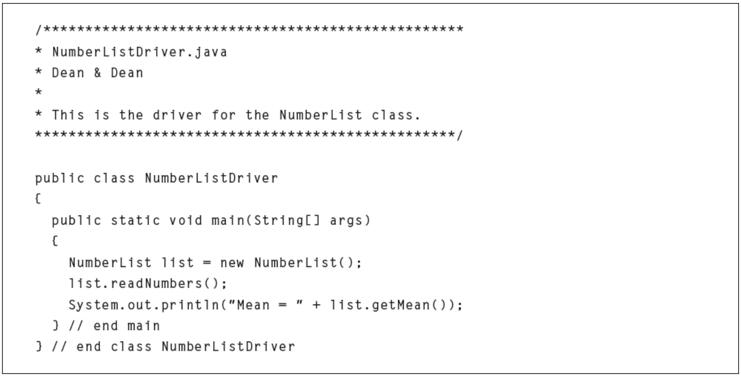
Unless you're incredibly careful, you've probably written programs that have generated runtime error messages. But prior to this chapter, you weren't properly prepared to understand those error messages thoroughly. Now you are. In this section, we describe exception messages by showing exception message details in the context of a complete program.

NumberList Program

Identify possible input errors.

The program in Figures 15.14a and 15.14b reads in a list of numbers and calculates the mean. The program compiles and runs successfully most of the time, but it's not very robust. There are three types of entries that make the program crash. We'll describe those three entry types, but before you read about them, first try to determine them on your own.

Page 715

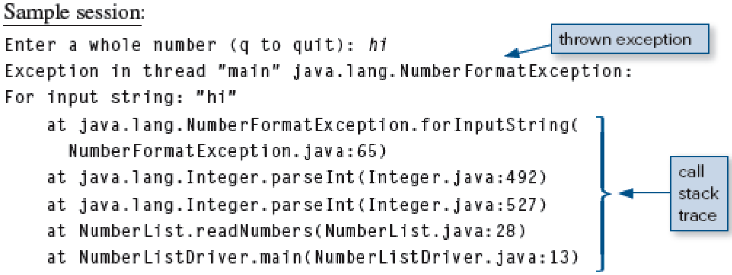


```
/*****************************************************
* NumberListDriver.java
* Dean & Dean
*
* This is the driver for the NumberList class.
*****************************************************/

public class NumberListDriver
{
  public static void main(String[] args)
  {
    NumberList list = new NumberList();
    list.readNumbers();
    System.out.println("Mean = " + list.getMean());
  } // end main
} // end class NumberListDriver
```

Figure 15.14a NumberList program driver that drives the class in Figure 15.14b

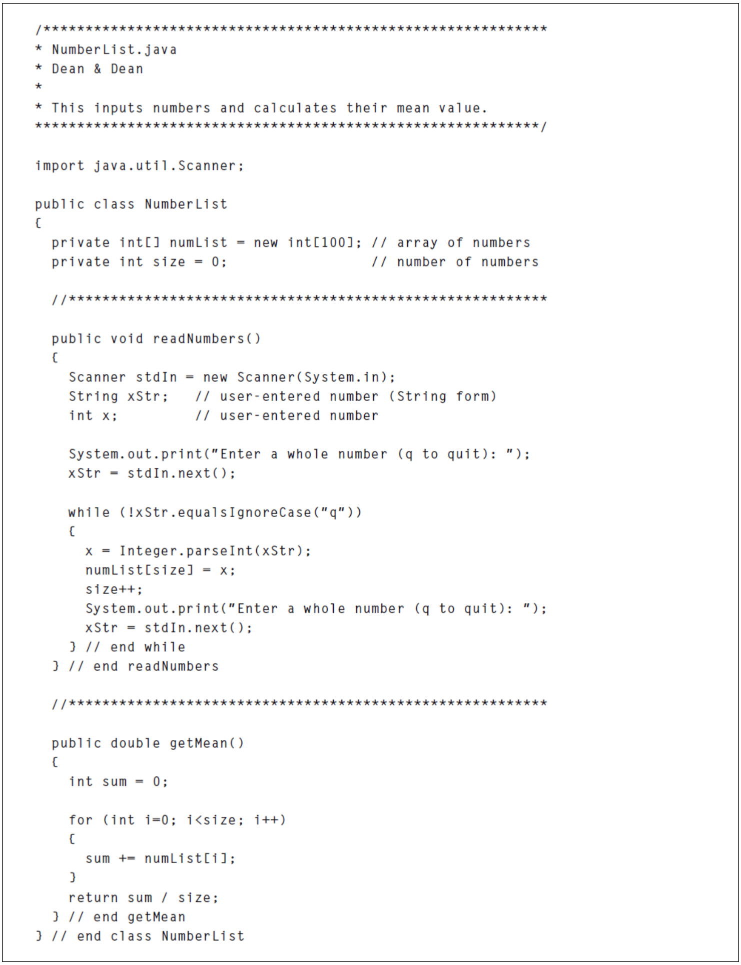
User Enters a Noninteger

In the readNumbers method, note the parseInt call. If the user enters a q, the while loop terminates ⚠ and parseInt is not called. But if the user enters something other than q, parseInt is called. If parseInt is called with a noninteger argument, then parseInt throws a NumberFormatException. And because there's no `try-catch` structure the JVM prints a detailed error message and then terminates the program. For example, if the user enters hi, the JVM prints a detailed error message and terminates the program, like this:[4]



Sample session:

```
Enter a whole number (q to quit): hi
Exception in thread "main" java.lang.NumberFormatException:
For input string: "hi"
    at java.lang.NumberFormatException.forInputString(
      NumberFormatException.java:65)
    at java.lang.Integer.parseInt(Integer.java:492)
    at java.lang.Integer.parseInt(Integer.java:527)
    at NumberList.readNumbers(NumberList.java:28)
    at NumberListDriver.main(NumberListDriver.java:13)
```



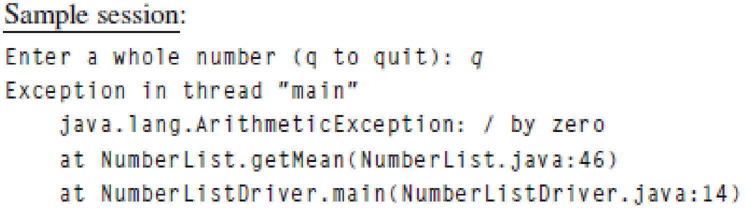
```
/*************************************************************
* NumberList.java
* Dean & Dean
*
* This inputs numbers and calculates their mean value.
*************************************************************/

import java.util.Scanner;

public class NumberList
{
  private int[] numList = new int[100]; // array of numbers
  private int size = 0;                 // number of numbers

  //***********************************************************

  public void readNumbers()
  {
    Scanner stdIn = new Scanner(System.in);
    String xStr;   // user-entered number (String form)
    int x;         // user-entered number

    System.out.print(”Enter a whole number (q to quit): ”);
    xStr = stdIn.next();

    while (!xStr.equalsIgnoreCase(”q”))
    {
      x = Integer.parseInt(xStr);
      numList[size] = x;
      size++;
      System.out.print(”Enter a whole number (q to quit): ”);
      xStr = stdIn.next();
    } // end while
  } // end readNumbers

  //***********************************************************

  public double getMean()
  {
    int sum = 0;

    for (int i=0; i<size; i++)
    {
      sum += numList[i];
    }
    return sum / size;
  } // end getMean
} // end class NumberList
```

Figure 15.14b NumberList class that computes the mean of input numbers

Let’s analyze the error message. First, the JVM prints the exception that was thrown. In this case, it is NumberFormatException. Then it prints a *call stack trace*. A call stack trace is a listing of the methods that were called prior to the crash, in reverse order. What methods were called? First main, then readNumbers, then parseInt. Note the numbers at the right side of the call stack trace. They are the line numbers in the source code for where the methods are called. For example, the 13 in the bottom line says that main's 13th line is a call to the readNumbers method.

User Immediately Enters q to Quit

At the bottom of the getMean method, note the division operation. Whenever you perform integer division, you should always be sure to avoid division by zero. In the NumberList program, it's not avoided. ⚠ The size instance variable is initialized to zero, and if the user immediately enters q to quit, size stays at zero and getMean performs division by zero. Integer division by zero throws an ArithmeticException. Because there's no `try-catch` mechanism, the JVM prints a detailed error message and terminates the program, like this:



Sample session:

```
Enter a whole number (q to quit): q
Exception in thread "main"
    java.lang.ArithmeticException: / by zero
    at NumberList.getMean(NumberList.java:46)
    at NumberListDriver.main(NumberListDriver.java:14)
```

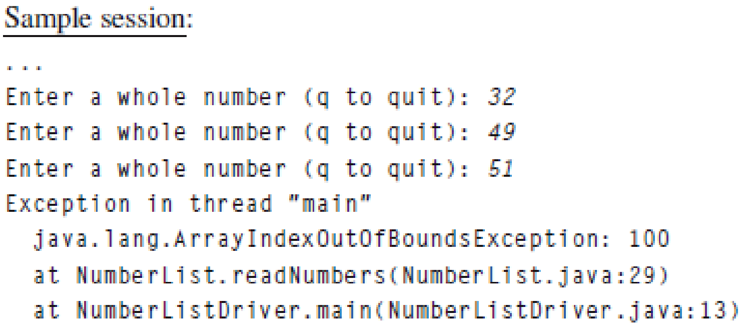
Note that if you perform floating-point division with a denominator of zero, there is no exception. If the numerator is a positive number, division by 0.0 returns the value Infinity. If the numerator is a negative number, division by 0.0 returns the value -Infinity. If the numerator is also 0.0, division by 0.0 returns the value NaN (NaN is a named constant in the Double class, and it stands for "not a number").

User Enters More Than 100 Numbers

In the NumberList program's instance-variable declarations, note that numList is a 100 element array. In the readNumbers method, note how this statement assigns user-entered numbers into the numList array:

```
numList[size] = x;
```

If the user enters 101 numbers, then the size variable increments to 100. That's bigger than the maximum index (99) in the instantiated array. If you access an array element with an index that's greater than the maximum index or less than zero, the operation throws an ArrayIndexOutOfBoundsException. Because ⚠ there are no try and catch blocks, the JVM prints a detailed error message and then terminates the program, like this:



Sample session:

```
. . .
Enter a whole number (q to quit): 32
Enter a whole number (q to quit): 49
Enter a whole number (q to quit): 51
Exception in thread "main"
  java.lang.ArrayIndexOutOfBoundsException: 100
  at NumberList.readNumbers(NumberList.java:29)
  at NumberListDriver.main(NumberListDriver.java:13)
```

We've now finished our description of the NumberList program's three runtime errors. Normally, when you see such errors, you should fix your code so as to avoid the runtime errors in the future. So for the NumberList program, you should add fixes for the three runtime errors. One of the chapter exercises asks you to do just that.

15.12 Using a throws Clause to Postpone the catch

In all the examples so far, we've

handled thrown exceptions locally; that is, we've put the try and catch blocks in the method that contains the dangerous statement. But sometimes that's not feasible.

Moving try and catch Blocks Back to the Calling Method

When it's not feasible to use local try and catch blocks, you can move the try and catch blocks out of the dangerous statement's method and back to the calling method. If you do that and the dangerous statement throws an exception, the JVM immediately jumps out of the dangerous statement's method and passes the exception back to the try and catch blocks in the calling method.[5]

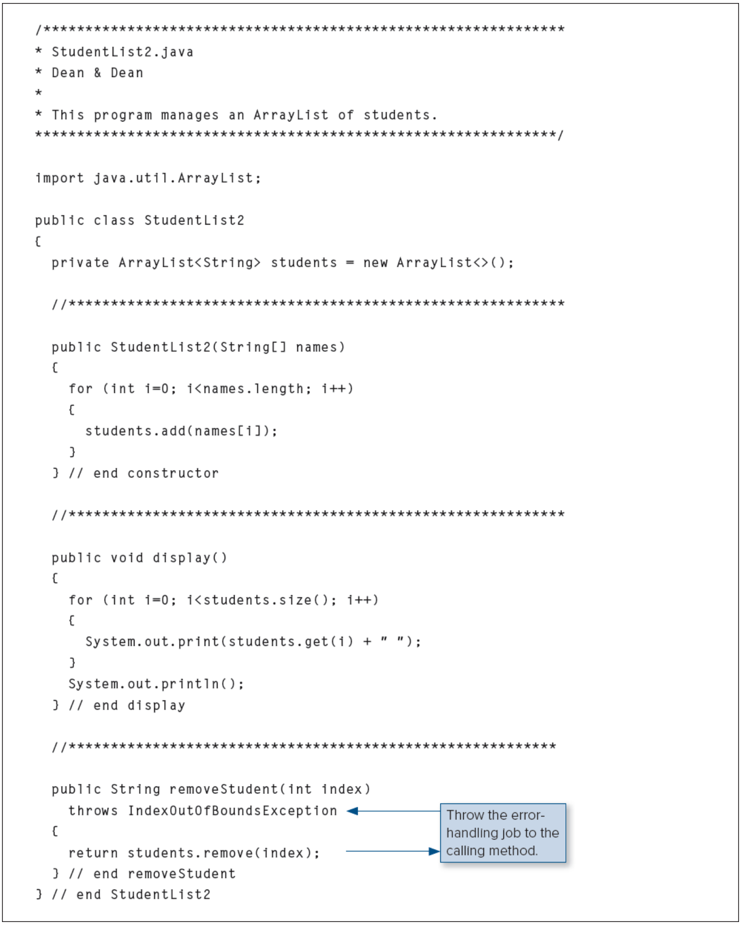
So when should you put try and catch blocks in the calling method, as opposed to in the dangerous statement's method? Most of the time, you should put your try and catch blocks in the dangerous statement's method because that promotes modularization, which is a good thing. But sometimes it's hard to come up with an appropriate catch block when you're inside the dangerous statement's method. For example, suppose that you've written a utility method that's called from lots of different places, and the method sometimes throws an exception. When an exception is thrown, you'd like to have an error message that's customized to the calling method. It's hard to do that if the catch block is in the utility method. The solution is to move the try and catch blocks to the calling methods.

Consider another example. Suppose you've written a method with a non-void return type that sometimes throws an exception. With a non-void return type, the compiler expects the method to return a value. But when an exception is thrown, you normally don't want to return a value because there's no appropriate value to return. So how can you have a non-void method and not return a value? Move the try and catch blocks to the calling method. Then when an exception is thrown, the JVM returns to the calling method without returning a value. The calling method's try and catch blocks handle the thrown exception, most likely with an error message. Let's see how this works in a Java program.

StudentList Program Revisited

Figure 15.15 contains a modified version of Figure 15.8's StudentList class. The main difference is that the removeStudent method now returns the name of the student it removes. This enables the calling method to do something with the removed element.

In the removeStudent method, note the return statement. The students.remove method call attempts to remove the element at the position indicated by index. If index is less than zero or greater than the index of the last element, then the JVM throws an IndexOutOfBoundsException. In our previous StudentList class, we handled the exception locally, within the removeStudent method. This time, because we're returning a value, it's more convenient to transfer the exception handling work back to the calling method. We do that by putting try and catch blocks in the calling method and by putting a throws clause in the removeStudent method's heading. Here's the heading:



```
/*************************************************************
* StudentList2.java
* Dean & Dean
*
* This program manages an ArrayList of students.
*************************************************************/

import java.util.ArrayList;

public class StudentList2
{
  private ArrayList<String> students = new ArrayList<>();

  //***********************************************************

  public StudentList2(String[] names)
  {
    for (int i=0; i<names.length; i++)
    {
      students.add(names[i]);
    }
  } // end constructor

  //***********************************************************

  public void display()
  {
    for (int i=0; i<students.size(); i++)
    {
      System.out.print(students.get(i) + " ");
    }
    System.out.println();
  } // end display

  //**********************************************************

  public String removeStudent(int index)
    throws IndexOutOfBoundsException
  {
    return students.remove(index);
  } // end removeStudent
} // end StudentList2
```

Throw the error-handling job to the calling method.

Figure 15.15 StudentList2 class, which is driven by the class in Figure 15.16

Page 720

```
public String removeStudent(int index)
  throws IndexOutOfBoundsException
```

Adding the throws clause reminds the compiler that the method might throw an unhandled exception. The throws clause is required if the unhandled exception is a checked exception, and it's just recommended if the unhandled exception is an unchecked exception. Because the IndexOutOfBoundsException is an unchecked exception, it's legal to omit the above throws clause. But it's good style to include it because it provides valuable self-documentation. If a programmer later wants to use the removeStudent method, the throws clause warns the programmer to provide a "remote" `try-catch` mechanism to handle the IndexOutOfBoundsException when calling removeStudent.

To see how to implement this "remote" `try-catch` mechanism, look at the StudentList2Driver class in Figure 15.16. It displays a list of students, asks the user which student should be removed, and attempts to remove that student. If the removeStudent method call throws an exception, it’s handled by the catch block in StudentList2Driver, which asks the user again which student should be removed.

Do you remember how we used throws Exception back in Section 3.24? We characterized that use as “quick and dirty” because it swept the compiler’s complaint “under the rug.” Are we doing that again here? No. This time, the calling method takes responsibility and deals with whatever exception might be thrown. As long as some caller eventually deals with thrown exceptions, it’s possible to postpone the catch more than once by rethrowing to a previous caller. However, to keep debugging manageable, you should avoid deep hierarchies of postponed catches because they make it more difficult to determine where a failure occurred. In particular, you should not postpone the catch if you make a recursive call

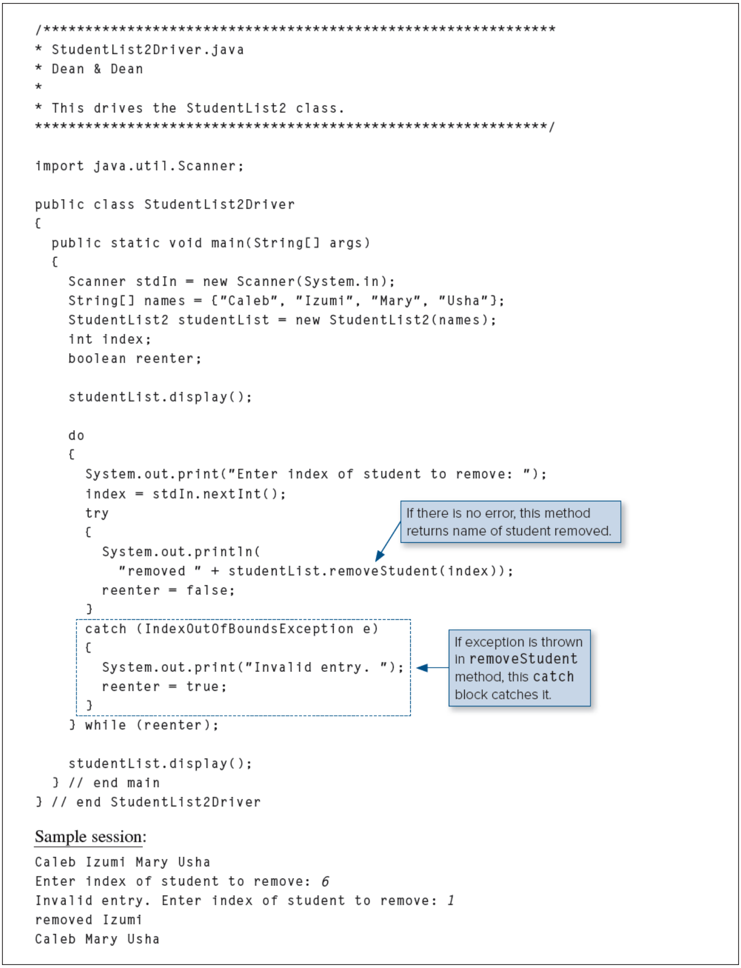
(Chapter 11) within a try block. The next section addresses another possible problem—idle resource accumulation—which can create problems regardless of whether the catch is postponed.

15.13 Automatic Cleanup Using Try-With-Resources

In implementing an exception handler, you’ll sometimes need to provide “cleanup code” that executes regardless of whether an exception is thrown. Although in principle that cleanup code could be anything, in practice, it’s almost always just the closing of a Closeable resource like a file handler. After writing to a file, you must *close* that file somehow to complete the writing process. Closing files also releases system resources and improves system performance.

Unfortunately, if some operation between file opening and file closing throws an exception, the close operation might be skipped. If a method throws exceptions back to a caller, the StudentList2 program in Figures 15.15 and 15.16 shows that normally you don't need a `try-catch` mechanism in the method throwing the exception. But if that method needs to close a file, you can put the file opening and file processing code inside a `try` block, and then follow that (otherwise unnecessary) try block with a separate finally block, which looks something like this:

```
finally
{
  if (fileOut != null)
  {
    fileOut.close();
  }
} // end finally
```



```
/*************************************************************
* StudentList2Driver.java
* Dean & Dean
*
* This drives the StudentList2 class.
*************************************************************/

import java.util.Scanner;

public class StudentList2Driver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String[] names = {"Caleb", "Izumi", "Mary", "Usha"};
    StudentList2 studentList = new StudentList2(names);
    int index;
    boolean reenter;

    studentList.display();

    do
    {
      System.out.print("Enter index of student to remove: ");
      index = stdIn.nextInt();
      try
      {
        System.out.println(
          "removed " + studentList.removeStudent(index));
        reenter = false;
      }
      catch (IndexOutOfBoundsException e)
      {
        System.out.print("Invalid entry. ");
        reenter = true;
      }
    } while (reenter);

    studentList.display();
  } // end main
} // end StudentList2Driver
```

If there is no error, this method returns name of student removed.

If exception is thrown in `removeStudent` method, this `catch` block catches it.

Sample session:

```
Caleb Izumi Mary Usha
Enter index of student to remove: 6
Invalid entry. Enter index of student to remove: 1
removed Izumi
Caleb Mary Usha
```

Figure 15.16 Driver for the StudentList2 class

Page 722

The finally block gets executed after the try block, regardless of whether the try block's code throws an exception. Thankfully, starting with Java 7, there's a better way to close a file.

All classes that implement Closeable also implement AutoCloseable. When an object is AutoCloseable, instead of closing it with an explicit close statement, we can ask the JVM to close it for us. We request this service by opening the AutoCloseable object in a *try-with-resources* header. This alternative approach simplifies code and makes programs more robust because Java automatically closes the resource (typically a file handler) regardless of whether all the code in the try-with-resources header and the following try block executes successfully or throws an exception.

In the try-with-resources approach, the creation of the AutoCloseable resource (both declaration and initialization) moves into a header in a set of parentheses after the keyword, try. When the `try-catch` mechanism is all in the same method, this avoids an explicit close statement. When the try-with-resources is in a method that throws its exceptions, this avoids a finally block. With semicolon separators between them, we can put any number of AutoCloseable resource-creation statements in the parentheses after the try, and the JVM closes all of them automatically at the end of that try block, regardless of whether exceptions are thrown.

The WriteToFile program in Figure 15.17 shows the case in which the try-with-resources is in a write method that throws an IOException. This is like StudentList2's removeStudent method that throws an IndexOutOfBoundsException. It's different in that WriteToFile's write method creates a PrintWriter file handler, which it must eventually close. Notice, however, that WriteToFile's write method contains no explicit close method call, and it contains no finally block. It avoids these by opening the file in a try-with-resources header.

The WriteToFile program in Figure 15.17 postpones the catch. However, try-with-resources also works when the catch block(s) are in the same method, immediately after the try. One of the exercises asks you to implement this alternative version of the WriteToFile program.

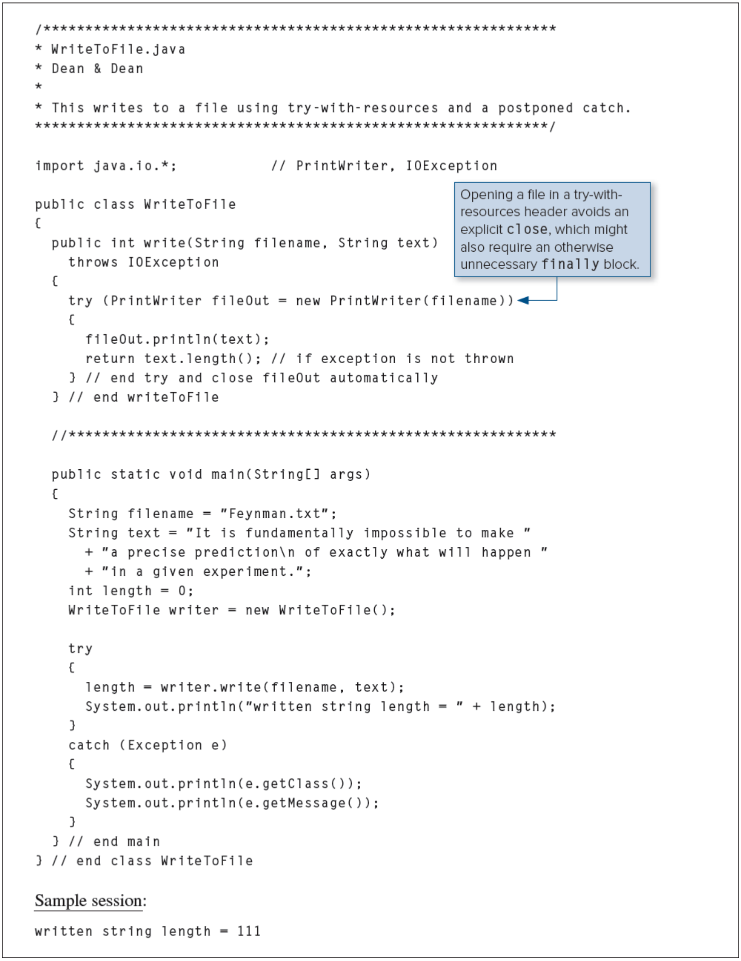
While the try-with-resources mechanism replaces the most common application of a finally block (releasing opened resources), the try-with-resources mechanism does not prevent you from also using a finally block for other cleanup activities. In addition, be aware that you can nest a `try-catch` mechanism inside a larger try block or inside a larger try-with-resources block. Similarly, you can nest a try-with-resources block and its associated catch block(s) inside a larger try block or inside a larger try-with-resources block. In other words, so long as every statement in a try-with-resources header opens something that is AutoCloseable, you can use a try-with-resources block just like an ordinary try block.

15.14 GUI Track: Line Plot Example Revisited (Optional)

Earlier in this chapter we implemented a LinePlot program that printed numerical values denoting start and end points of the straight sections between points on a line plot. For example, here is what that program produces for a five-segment line that goes from point (0,0) to (1,3) to (2,1) to (3,2) to (4,2) to (5,1):

```
Enter x & y coordinates (q to quit): 1 3
New segment = (0,0)-(1,3)
Enter x & y coordinates (q to quit): 2 1
New segment = (1,3)-(2,1)
Enter x & y coordinates (q to quit): 3 2
New segment = (2,1)-(3,2)
Enter x & y coordinates (q to quit): 4 2
```

Page 723



```
/*************************************************************
* WriteToFile.java
* Dean & Dean
*
* This writes to a file using try-with-resources and a postponed catch.
*************************************************************/

import java.io.*;           // PrintWriter, IOException

public class WriteToFile
{
  public int write(String filename, String text)
    throws IOException
  {
    try (PrintWriter fileOut = new PrintWriter(filename))
    {
      fileOut.println(text);
      return text.length(); // if exception is not thrown
    } // end try and close fileOut automatically
  } // end writeToFile

  //***********************************************************

  public static void main(String[] args)
  {
    String filename = ”Feynman.txt”;
    String text = ”It is fundamentally impossible to make ”
      + ”a precise prediction\n of exactly what will happen ”
      + ”in a given experiment.”;
    int length = 0;
    WriteToFile writer = new WriteToFile();

    try
    {
      length = writer.write(filename, text);
      System.out.println(”written string length = ” + length);
    }
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end class WriteToFile
```

Opening a file in a try-with-resources header avoids an explicit `close`, which might also require an otherwise unnecessary `finally` block.

Sample session:

```
written string length = 111
```

Figure 15.17 WriteToFile program,

which uses try-with-resources and automatic closing

```
New segment = (3,2)-(4,2)
Enter x & y coordinates (q to quit): 5 1
New segment = (4,2)-(5,1)
Enter x & y coordinates (q to quit): q
```

The output is understandable, but kinda boring in terms of presentation. Instead of plain text for output, let's use JavaFX to generate a graphical presentation, like that in Figure 15.18. The plotted points—(1,3), (2,1), and so on—come from GUI user inputs.

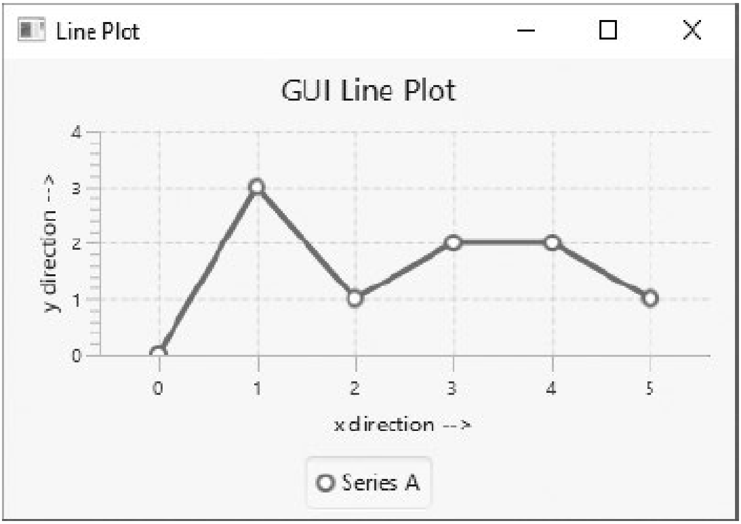

Line Plot
GUI Line Plot
y direction -->
0
1
2
3
4
0
1
2
3
4
5
x direction -->
Series A

Figure 15.18 Sample output for the LinePlotGUI program

©JavaFX

Figure 15.19a shows the first part of the program. If you've read the preceding chapters' optional GUI sections, you'll recognize the first five import statements. The sixth import statement is our old friend the Integer wrapper class. The final import statement provides access to three new JavaFX classes that this section's plotting application needs. The two instance constants specify the stage's size. The instance variable yCoords is an array to hold user-input y-coordinates.

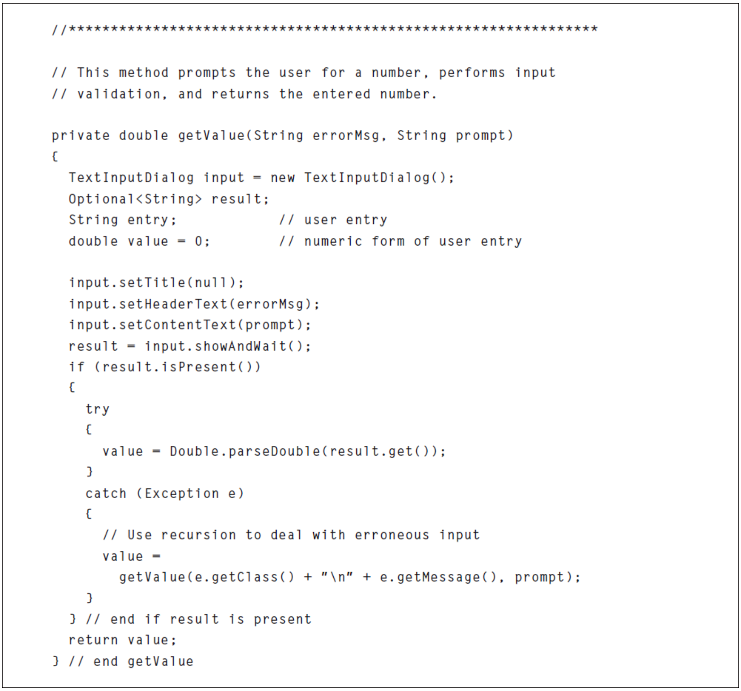
In the start method, we initialize the local variable numOfPoints with the value returned by a getValue helper method. Later you'll see how the getValue method deals with erroneous nonnumeric entries. The start method uses an input validation loop to deal with a negative or zero entry for number of points. The (int) cast operator rounds off a possible decimal entry to obtain an integral number of points. The statement after the do loop initializes the instance variable yCoords with a reference to a new array having a length equal to a legitimate numOfPoints. The subsequent for loop iterates through that array and fills it with user-supplied values for each point's y-coordinate. The last three statements set a title, set the scene with a GUI LineChart returned by the other helper method, getChart, and show that chart on the computer screen.

Figure 15.19b shows the getValue helper method, which collects user inputs for the number of points and the y-coordinate values. It does this by using a JavaFX TextInputDialog to get input from the user in the form of a string. It then attempts to convert that input string to a double value. The getValue method employs a `try-`

`catch` mechanism to check for input that is not convertible to a double. In the catch block, it handles errant input by recursively calling itself with an error message argument that displays in the TextInputDialog’s header section. The method call has a second argument for the prompt, and the prompt displays in the TextInputDialog’s content section. When the user enters valid input the method returns that value.

```
/******************************************************************
* LinePlotGUI.java
* Dean & Dean
*
* This draws a line between double values at discrete points.
******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.TextInputDialog;
import java.util.Optional;
import java.lang.Integer;
import javafx.scene.chart.*; // LineChart, CategoryAxis, NumberAxis

public class LinePlotGUI extends Application
{
  private final int WIDTH = 400;
  private final int HEIGHT = 250;
  private double[] yCoords;  // y coordinate values

  //***************************************************************

  public void start(Stage stage)
  {
    int numOfPoints;

    do
    {
      numOfPoints=
        (int) getValue(null, ”Enter positive number of points: ”);
    } while (numOfPoints <= 0);
    yCoords = new double[numOfPoints];
    for (int i=0; i<numOfPoints; i++)
    {
      yCoords[i] =
        getValue(null, ”At x = ” + i + ”, what is y value?” +
          ”\nEnter an integer or decimal number:”);
    } // end for
    stage.setTitle(”Line Plot”);
    stage.setScene(new Scene(getChart(), WIDTH, HEIGHT));
    stage.show();
  } // end start
```

Figure 15.19a LinePlotGUI class—part A



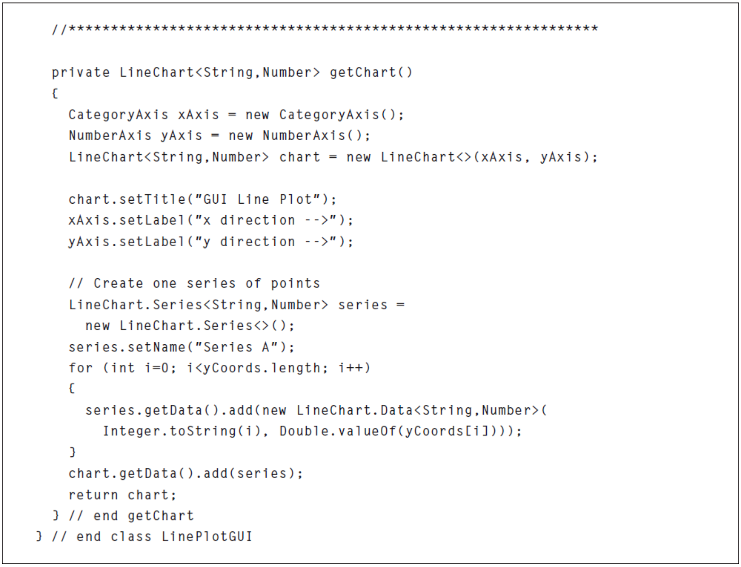
```
//*****************************************************************

// This method prompts the user for a number, performs input
// validation, and returns the entered number.

private double getValue(String errorMsg, String prompt)
{
  TextInputDialog input = new TextInputDialog();
  Optional<String> result;
  String entry;            // user entry
  double value = 0;        // numeric form of user entry

  input.setTitle(null);
  input.setHeaderText(errorMsg);
  input.setContentText(prompt);
  result = input.showAndWait();
  if (result.isPresent())
  {
    try
    {
      value = Double.parseDouble(result.get());
    }
    catch (Exception e)
    {
      // Use recursion to deal with erroneous input
      value =
        getValue(e.getClass() + "\n" + e.getMessage(), prompt);
    }
  } // end if result is present
  return value;
} // end getValue
```

Figure 15.19b LinePlotGUI class—part B

Figure 15.19c contains the getChart method that constructs and returns the desired line plot. Note the statement that declares and initializes the chart reference variable:

LineChart<String,Number> chart = new LineChart<>(xAxis, yAxis);

LineChart is a class from JavaFX's javafx.scene.chart package. It's a generic class with type parameters that specify the types for the label values along the line chart's x-axis and y-axis. For the chart variable's declaration, we use String and Number for the x-axis and y-axis type parameters, respectively. For the LineChart object instantiation, we use xAxis and yAxis for the constructor arguments, where xAxis is a CategoryAxis object and yAxis is a NumberAxis object. As you can see from the output in Figure 15.18, we display numbers for both axes. So Number and NumberAxis make sense for the y-axis, but why String and CategoryAxis for the x-axis? In the program's output, can you see the tick marks between the integers along the y-axis? Showing the tick marks is the default for LineChart axes that are labeled with numbers. That's a good thing for the y-axis because the y values can be real numbers. But showing such tick marks for the x-axis would be misleading because the points' x values are for whole numbers (if the user enters 6, that means 6 points; entering 6.5 doesn't make sense).



```
  //***************************************************************

  private LineChart<String,Number> getChart()
  {
    CategoryAxis xAxis = new CategoryAxis();
    NumberAxis yAxis = new NumberAxis();
    LineChart<String,Number> chart = new LineChart<>(xAxis, yAxis);

    chart.setTitle("GUI Line Plot");
    xAxis.setLabel("x direction -->");
    yAxis.setLabel("y direction -->");

    // Create one series of points
    LineChart.Series<String,Number> series =
      new LineChart.Series<>();
    series.setName("Series A");
    for (int i=0; i<yCoords.length; i++)
    {
      series.getData().add(new LineChart.Data<String,Number>(
        Integer.toString(i), Double.valueOf(yCoords[i])));
    }
    chart.getData().add(series);
    return chart;
  } // end getChart
} // end class LinePlotGUI
```

Figure 15.19c LinePlotGUI class—part C

To avoid those intermediate tick marks on the x-axis, we could use Number and NumberAxis for the x-axis and then call the setTickUnit method with a value of 1 (setting the distance between tick marks to 1). As an alternative, we use String and

CategoryAxis for the x-axis. Categories are discrete and thus the CategoryAxis class is for when you want discrete values for your axis. Discrete values means no intermediate tick marks.

Continuing with our analysis of the getChart method, after chart gets initialized, we assign a title for the line chart and labels for the chart's two axes. The next two statements create a LineChart.Series object, which is a set of connected points. One line chart can display sets of points with multiple Series objects, but this simple example has only one. The for loop populates our one series with data points. The only extra code needed to avoid those pesky tick marks on the x-axis is in the first argument of the LineChart.Data constructor. Instead of being just i, that argument is Integer.toString(i).

After adding our data points to the Series object, we add the object to chart. Then we return the fully composed chart to the Scene constructor in the in the start method.

Summary

- An exception is an event that occurs during the execution of a program that disrupts the program's normal flow of instructions.
- Exception handling is a technique for handling exceptions gracefully.
- Use a `try` block to "try out" one or more dangerous method calls. If there's a problem with the dangerous method calls, the JVM throws an exception and looks for a "matching" catch block.
- A catch block is matching if the catch heading's parameter type is

the same as or an ancestor of the type of the thrown exception.

- If an exception is thrown, the JVM immediately jumps out of the current try block. That means that if there are statements in the try block after the exception-throwing statement, those statements get skipped.

- Checked exceptions must be checked with a `try-catch` mechanism.

- Unchecked exceptions may optionally be checked with a `try-catch` mechanism, but it's not a requirement.

- Unchecked exceptions are descendants of the RuntimeException class.

- To implement a simple, general-purpose exception handler, define a catch block with an Exception type parameter, and inside the catch block, call the Exception class's getClass and getMessage methods.

- To define an exception handler with more specificity, define a sequence of catch blocks. Use multiple exception catch parameters for common responses. Arrange the catch blocks with the more general exception classes later, and end with the general-purpose exception handler.

- If a program crashes, the JVM prints a call stack trace. A call stack trace is a listing of the methods called prior to the crash, in reverse order.

- When objects like a file handler are AutoCloseable, declare and initialize them (open the files) in a try-with-resources header. A single header may contain multiple opening statements, separated by semicolons. Then the JVM automatically closes all opened objects without an explicit close call, regardless of whether operations in the try-with-resources header or the associated try block throw exceptions.

- To propagate an exception back to the calling module, append `throws` *exception-type* to the called method header. Then the called method does not need try and catch blocks, but it may include a try-with-resources block and/or a finally block for cleanup.

Review Questions

§15.3 Using try and catch Blocks to Handle "Dangerous" Method Calls

1. If your program contains an API method call, you should put it inside a try block. To be fully compliant with proper coding practice, you should apply this rule for all your API method calls. (T / F)

2. A try block and its associated catch block(s) must be contiguous. (T / F)

§15.5 try Block Details

Page 729

3. Usually, you should try to aggregate related dangerous statements in the same try block to minimize clutter. (T / F)

4. Where should you put safe statements that use the results of dangerous operations?

5. If an exception is thrown, the JVM jumps to a matching catch block, and after executing the catch block, it returns to the try block at the point where the exception was thrown. (T / F)

6. In checking for compile-time errors, the compiler takes into account that all statements inside a try block might get skipped. (T / F)

§15.6 Two Categories of Exceptions—Checked and Unchecked

7. If an exception is derived from the RuntimeException class it is a(n) ____________________ exception.

8. Checked exceptions are exceptions that are in or derived from the ____________________ class, but not in or derived from the ____________________ class.

§15.7 Unchecked Exceptions

9. In the following list, indicate

whether each option is a viable option for an unchecked exception that you know your program might throw:

a) Ignore it.

b) Rewrite the code so that the exception never occurs.

c) Put it in a try block, and catch it in a following catch block.

§15.8 Checked Exceptions

10. When a statement might throw a checked exception, you can keep the compiler from complaining if you put that statement in a try block and follow the try block with a catch block whose parameter type is the same as the exception type. (T / F)

11. You can determine whether a particular statement contains a checked exception and the type of that exception by attempting to compile with no `try-catch` mechanism. (T / F)

§15.9 Generic catch Block with Exception Class

12. Is it OK to include code that can throw both unchecked and checked exceptions in the same try block?

13. What type of exception matches all checked exceptions and all unchecked exceptions except those derived from the Error class?

14. What does the getMessage method return?

§15.10 Multiple catch Blocks and Multiple Exceptions per Block

15. The compiler automatically checks for out-of-order catch blocks. (T / F)

16. Write the header for a catch block that catches either an InvalidPathException or a NoSuchFileException.

§15.11 Understanding Exception Messages

17. What are the two types of information displayed by the JVM when it encounters a runtime error that terminates execution?

§15.12 Using a throws Clause to Postpone the catch

18. Suppose you want to postpone catching of a NumberFormatException. What should you append to the heading of a method to alert the compiler and a potential user that something in the method might throw that type of exception?

19. Given a non-void method that contains no try and catch blocks. If the method throws an exception, we know that the JVM transfers the thrown exception back to the calling method. But does the JVM return a value (with a return statement) to the calling module?

§15.13 Automatic Cleanup Using Try-With-Resources

20. The JVM will close your file automatically if you instantiate its file handler in a try-with-resources header, like this (T / F):

```
public void writeToFile(String filename, String text)
  throws IOException
{
  PrintWriter fileOut;
  try (fileOut = new PrintWriter(new File(filename)))
  {
    fileOut.println(text);
  }
} // end writeToFile
```

Exercises

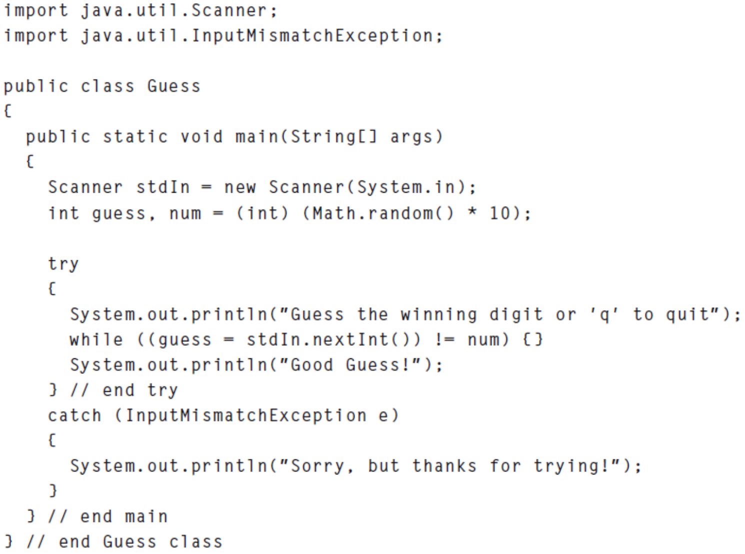
1. [after §15.3] Given the program below, what is the output if the user enters 3 in response to the prompt?

```
import java.util.Scanner;

public class Athletes
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String[] players = {”Serena Williams”, ”Alex Morgan”,
      ” Allyson Felix”, ”Lindsey Vonn”, ”Katie Ledecky”};
    String ranking;
    int index = 0;

    System.out.print(“Enter ranking: ”);
    ranking = stdIn.nextLine();
    try
    {
      index = Integer.parseInt(ranking);
      System.out.println(”Rank = ” + index);
    }
    catch (NumberFormatException e)
    {
      System.out.println(“Entered value must be an integer.”);
    }
    try
    {
      System.out.println(players[index - 1]);
    }
    catch (IndexOutOfBoundsException e)
    {
      System.out.println(“Entry must be between 1 and ”
        + players.length);
    }
     System.out.println(”done”);
  } // end main
} // end class Athletes
```

2. [after §15.3] For the program

in Exercise 1, what is the output if the user enters “first” in response to the prompt?

3. [after §15.5] Explain what the following program does for different possible user inputs.



```
import java.util.Scanner;
import java.util.InputMismatchException;

public class Guess
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int guess, num = (int) (Math.random() * 10);

    try
    {
      System.out.println(”Guess the winning digit or ’q’ to quit”);
      while ((guess = stdIn.nextInt()) != num) {}
      System.out.println(”Good Guess!”);
    } // end try
    catch (InputMismatchException e)
    {
      System.out.println(”Sorry, but thanks for trying!”);
    }
  } // end main
} // end Guess class
```

4. [after §15.7] Suppose you have a program with a method call you’re unfamiliar with. Describe the best way to determine if that method might throw an unchecked exception. And if you find that the method call might call an unchecked exception, what are the strategies for dealing with that situation?

5. [after §15.9] Early in the development of a new program that is likely to throw exceptions of many types, what can you do to enable early testing to help identify types of exceptions that might be thrown and why they are thrown? Specifically, provide a catch block that catches any thrown exception, prints a message that identifies the thrown exception, and prints a message that describes what caused the thrown exception.

6. [after §15.10] Multiple catch Blocks:

Suppose the code in a try block might throw any of the following exceptions:

IOException

NoSuchElementException

Exception

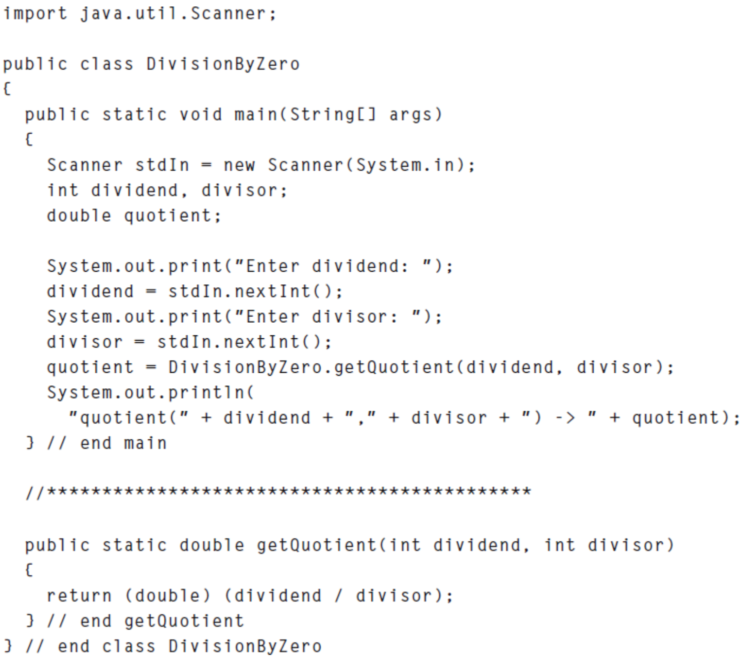
InputMismatchException

RuntimeException

Identify an acceptable sequence for the given exceptions if they were in a sequence of catch block headings.

7. [after §15.11] Handling an ArithmeticException.

The following program calls getQuotient, which divides its two passed-in values and returns the quotient as a floating-point number.



```
import java.util.Scanner;

public class DivisionByZero
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int dividend, divisor;
    double quotient;

    System.out.print(”Enter dividend: ”);
    dividend = stdIn.nextInt();
    System.out.print(”Enter divisor: ”);
    divisor = stdIn.nextInt();
    quotient = DivisionByZero.getQuotient(dividend, divisor);
    System.out.println(
      ”quotient(” + dividend + ”,” + divisor + ”) -> ” + quotient);
  } // end main

  //*******************************************

  public static double getQuotient(int dividend, int divisor)
  {
    return (double) (dividend / divisor);
  } // end getQuotient
} // end class DivisionByZero
```

The program works well for the normal cases, but when the user enters 0 for the divisor, the program generates this error:

> Exception in thread "main" java.lang.ArithmeticException: / by zero

Modify the getQuotient method by putting the division operation in a try block and following it with a catch block that returns either Double.POSTIVE_INFINITY if the dividend is positive or Double.NEGATIVE_INFINITY if the dividend is negative.

8. [after §15.11] Correcting Problems:

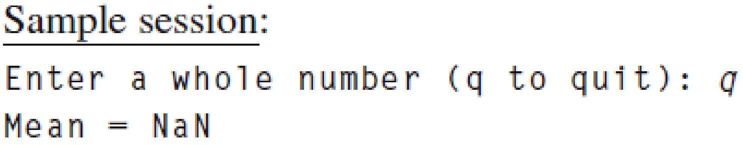
Fix the problems in the NumberList program without making any changes to the NumberListDriver class.

a) Avoid the possibility of an ArrayIndexOutOfBoundsException by adding to the while condition, size < numList.length, and perform the query and entry at the end of the while loop only if size < numList.length.

b) If the entry is not a "q" and if it is not a legal integer, catch the exception, and in the catch block use the getClass method inherited from the Object class to print the name of the exception class followed by the error message with the statement:

```
System.out.println(e.getClass()
+ " " + e.getMessage());
```

c) If a user immediately enters “q” to quit, print “NaN” by making a small program correction that utilizes double’s NaN value, and avoids using the try–catch mechanism to catch the int arithmetic exception.

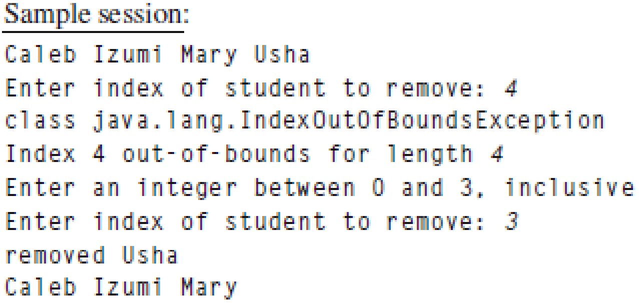


Sample session:

```
Enter a whole number (q to quit): q
Mean = NaN
```

9. [after §15.12] Postponed catch elaboration:

Modify this chapter’s StudentList2 program so that the driver’s exception handler displays more than just “Invalid entry.” Instead, have it display the error class and message and the acceptable range of index values. Most of the change will be in the driver class’s catch block, but in your StudentList2 class you’ll want to add a size method that returns the size of the students ArrayList.



Sample session:

```
Caleb Izumi Mary Usha
Enter index of student to remove: 4
class java.lang.IndexOutOfBoundsException
Index 4 out-of-bounds for length 4
Enter an integer between 0 and 3, inclusive
Enter index of student to remove: 3
removed Usha
Caleb Izumi Mary
```

10. [after §15.13] Simple try-with-resources:

Implement a program named WriteToFileEx that writes and displays what this chapter’s WriteToFile program writes and displays. But do not use a separate write method. Put the try-with-resources block together with the catch block in the main method.

Review Question

Solutions

1. False. Many API method calls are safe, and there's no need to put those method calls inside a try block.

2. True. You cannot put any statements between associated try and catch blocks.

3. True.

4. Put safe statements that use the results of dangerous operations inside the try block and after those dangerous operations.

5. False. After executing the catch block, the JVM continues downward; it does not jump back to the try block. Consequently, try-block statements get skipped if they follow an exception-throwing statement.

6. True.

7. If an exception is derived from

the RuntimeException class, it is an unchecked exception.

8. Checked exceptions are exceptions that are in or derived from the Exception class, but not in or derived from the RuntimeException class.

9. Viable options for an unchecked exception that you know might be thrown:

- **a)** Not viable! You don't want your program to crash at runtime.
- **b)** Viable.
- **c)** Viable.

10. True.

11. True. If the statement contains a checked exception, the compiler will say so and identify the exception type.

12. Yes.

13. The Exception exception.

14. The Exception class's

getMessage method returns a text description of the thrown exception.

Page 734

15. True. The compiler complains if an earlier more generic catch block preempts a later more specific catch block.

16. `catch (InvalidPathException | NoSuchFileException e)`

17. The two types of information displayed by the JVM when it encounters a runtime error are:

- **a)** Identification of the particular exception thrown.
- **b)** A call-stack trace, which is a reverse-order listing of the methods called just prior to the crash, along with the line numbers where the error occurred in each method.

18. You must append throws

NumberFormatException to the end of the method heading.

19. No. When an exception is thrown back to the calling method, the JVM does not return a value (with a return statement) to the calling module.

20. False. The try-with-resources header must also include the file handler's declaration.

---

[1] "The JavaTM Tutorials," accessed July 9, 2019, https://docs.oracle.com/javase/tutorial/essential/exceptions/definition.html.

[2] Actually, as you'll see in Section 15.9, a catch block is also considered to be matching if the catch heading's parameter type is a superclass of the thrown exception's class.

[3] For more information, you can call the printStackTrace method. This generates voluminous error output like

that described later in Section 15.11.

[4] The formatting of the error message may be slightly different, but the information will be similar. In a catch block, you could generate a message like this intentionally by calling Exception's printStackTrace method.

[5] Actually, the jump to the calling method is not immediate if there's a finally block below the try block(s). In that case, the JVM jumps to the finally block prior to jumping to the calling method. We describe the finally block in the next section.

Files, Buffers, Channels, and Paths

Objectives

- Copy simple text from and to small local files.
- Learn the rudiments of HTML.
- Copy HTML from a remote website.

- Learn how to copy Java objects to and from Java object files.
- Be able to use different character sets and different file-opening options.
- Write and read large text files efficiently through text buffers.
- Copy primitive data to or from any place in a primitive buffer.
- Copy bytes randomly to or from any place in a file through a Java `Channel`.
- Establish a nonvolatile connection to file data with a memory map.
- Learn how to manipulate paths.

Outline

16.1 Introduction 736

16.2 Simple Text-File Example:

HTML File Generator 737

16.3 A Website Reader 741

16.4 Object File I/O 743

16.5 Character Sets and File-Access Options 748

16.6 Buffered Text File I/O 749

16.7 Primitive Buffers with Random Access 752

16.8 Channel I/O and Memory-Mapped Files 760

16.9 Path, Whole-File, and Directory Operations 767

16.10 Walking a Directory Tree 769

16.11 GUI Track: Final Iteration of Problem Solving with CRC Cards (Optional) 775

16.1 Introduction

Except for the simple ReadFromFile and WriteToFile programs in the previous chapter, up until now, program input has come from the keyboard and program output has gone to the computer screen. That type of input/output (I/O) is temporary. When you enter input from the keyboard, the input is not saved. If you want to run the program again, you must enter the input again. Likewise, when you send output to the computer screen, it's not saved. If you want to analyze the output, you must analyze it on the screen. A day later, if you want to look at it again, you must run the program again.

For permanent or reusable I/O, you can store input or output data in a file. A *file* is a group of data that is typically stored in a contiguous block on a nonvolatile storage device (such as a hard disk). Data files are fundamentally the same as the `.java` and `.class` program files that you have been using all along to hold your Java programs. But instead of holding programs, data files hold data that programs read from for input or write to for output. For your data files, use an extension that suggests how the data is formatted and the type of program that understands that format. For example, `.txt` identifies a text file that a simple word-processing program can understand.

This chapter begins with Hypertext Markup Language (HTML) writing and reading. The HTML file-generator example reviews the simpler exception-handling techniques introduced in the previous chapter in the context of HTML text. This example opens two local files in a single try-with-resources header. The website-reader example in Section 16.3 opens a connection with a remote website. This shows how web access relates to file handling. Text files are easy to understand, and you can create or view text files with almost any text editor.

Section 16.4 shows how you can use software already in the Java language to perform the intricate structural conversion between a program object and the stream of bytes that flow to and from a file. Java's built-in software greatly simplifies the custom code needed to write objects to a file and read objects from a file. You'll learn how to write more than one distinct object to the same file and how to write modifications of previously written objects. And you'll learn how to read back any or all of those objects. Assuming the code that reads is consistent with the code that writes, and you know exactly how things are organized, object files are easy to use.

Section 16.5 will show you how to specify particular character sets to support different languages and how to control the way opened files may be manipulated. Section 16.6 shows how to transfer large blocks of text with maximum efficiency.

Section 16.7 shows how to convert each of the different types of primitive data or arrays of any of these types to the stream of bytes that flow to and from a file. It also shows how to access or mutate primitive data at random locations in one of those streams of bytes. Although it's more tedious than working with text or using Java's automatic processing of large objects, working directly with primitive data, and especially bytes, tends to improve a program's efficiency.

Section 16.8 extends these techniques to a larger scale and shows how to connect the streams of bytes to files. Memory mapping establishes a "permanent" link between your program and a particular file's data—a link that survives normal file closing.

Section 16.9 explains how you can specify and manipulate paths to directories and files; it explains how to copy or move files from one directory to another; and it shows how to display what's in a directory. Section 16.10 shows how to search the entire directory structure for files that have specified features in their names. And Section 16.11, which is optional, shows how to display a directory's contents with a graphical user interface.

This chapter focuses on transferring data to and from a local file, which directly or indirectly involves conversion to or from a stream of bytes. But converting program data to or from a stream of bytes has broader applications. If you want to transfer data to or from other computers through local networks or through the Internet, you also need to make the conversion between your program's data types and streams of bytes. This chapter shows you how to make those conversions. So while you are learning about data storage in files, you are also learning about data transfer through communication channels.

16.2 Simple Text-File Example: HTML File Generator

Readers who want to use file I/O early have the option of reading this section after completing Section 3.23 in Chapter 3. If you jump from Chapter 3 to here, you should be aware that some of the material in this section won’t make sense. But if you treat the file I/O code in the program in Figures 16.1a and 16.1b as a recipe, it will show you how to read from a file anything you can read from the keyboard, and it will show you how to write to a file anything you can print to the computer screen. This section consolidates what we explained in Chapter 15 about simple text file I/O with an example that reads from a text file and writes to a text file and opens both files in the same try-with-resources header. The program translates a user-specified text file into web page format. Then it writes the translation to a newly generated HTML file.

In Figure 16.1a, the program starts by reading a user-specified filename into the `filenameIn` variable. Then it composes the output file name. The name of the output file should be the same as the name of the input file except for the extension, which should be `.html`. To compose the name of the output file, the `String` method `lastIndexOf` finds the index of the last dot in `filenameIn`. If there is no dot, the `lastIndexOf` method returns a value of `-1`, and the program simply appends `.html` to the original filename. If there is a dot, `String`'s `substring` method returns the part of the string up through the character immediately before the dot, and the program adds `.html` to that. This process replaces the original filename's extension with a `.html` extension, and it assigns the result to `filenameOut`.

In the try-with-resources header, the program uses `filenameIn` to open the input file and create a `Scanner` object called `fileIn` to handle file-reading operations. Also in the try-with-resources header, it uses `filenameOut` to open the output file and create a `PrintWriter` object called `fileOut` to handle file-writing operations. Both `fileIn` and `fileOut` close automatically when the `try` block ends. Within the `try` block, the code starts by checking for an empty input file. If it's empty, it prints a warning message. Otherwise, it proceeds to transform the input file's data into the output file's format.

To help you understand this transformation, we need to digress and provide a brief overview of HTML (the computer language used to create Internet web pages). This book is not about HTML, but it's worth learning a little about HTML because Internet web pages are what got the Java language going.

HTML overview:

- At the top of every web page, you should include a doctype declaration, which tells the web browser the language the page is written in. For most web pages, the language will be *html,* and the doctype declaration will be `<!DOCTYPE html>`.
- HTML *tags* are surrounded by

angled brackets, and they describe the purpose of their associated text.

- <html> and </html> tags surround the entire web page.
- The content between <head> and </head> tags is the heading for an HTML page. The heading contains information that describes the HTML page. This information is used by the browser and by search engines, but it is not visible on the HTML page.
- `<title>` and `</title>` tags surround the text that appears in a web page’s title bar. Internet search engines use the `<title>` content to find web pages.
- The content between `<body>` and `</body>` tags is the body for the HTML page. The body contains the text that’s displayed on the HTML page.

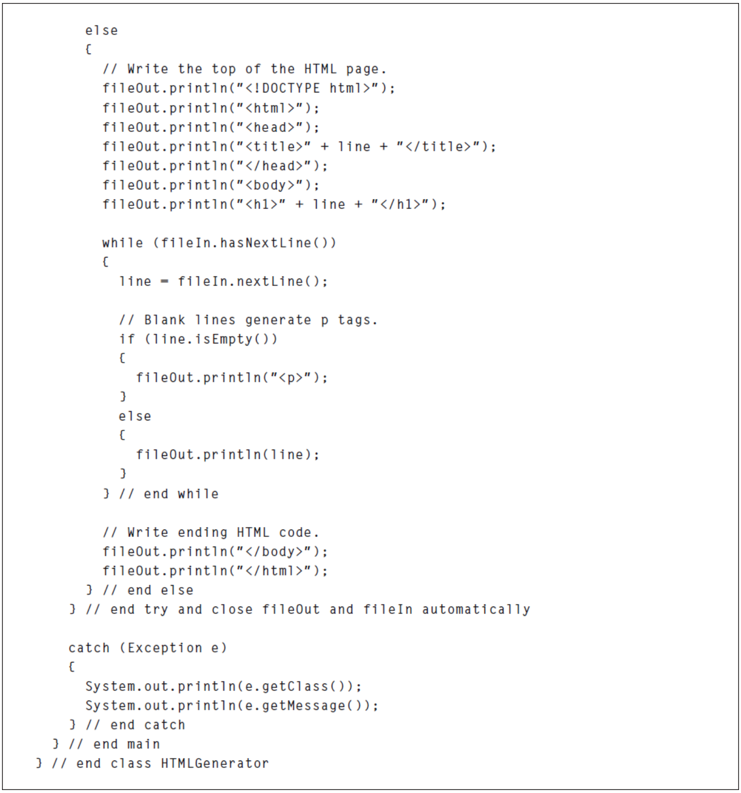
- `<h1>` and `</h1>` tags surround the text that appears as a heading within a web page. Web browsers use large fonts to display text that's surrounded by `<h1>` tags.
- `<p>` tags indicate the beginning of a new paragraph. Web browsers generate a blank line for each `<p>` tag, and this helps set paragraphs apart.

```
/*************************************************************
* HTMLGenerator.java
* Dean & Dean
*
* This program reads the text in a user-specified file and
* uses that text to generate an HTML file web page.
*************************************************************/

import java.util.Scanner;
import java.io.PrintWriter;
import java.nio.file.Paths;

public class HTMLGenerator
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String filenameIn;          // original file's name
    int dotIndex;               // position of dot in filename
    String filenameOut;         // HTML file's name
    String line;                // a line from the input file

    System.out.print("Enter file's name: ");
    filenameIn = stdIn.nextLine();

    // Compose the new filename
    dotIndex = filenameIn.lastIndexOf(".");
    if (dotIndex == -1)         // no dot found
    {
      filenameOut = filenameIn + ".html";
    }
    else                        // dot found
    {
      filenameOut =
        filenameIn.substring(0, dotIndex) + ".html";
    }

    try (
      Scanner fileIn = new Scanner(Paths.get(filenameIn));
      PrintWriter fileOut = new PrintWriter(filenameOut))
    {
      // First line used for title and header elements
      line = fileIn.nextLine();
      if (line == null)
      {
        System.out.println(filenameIn + " is empty.");
      }
```

This opens an input file and an output file.

Figure 16.1a HTMLGenerator program—part A

```
      else
      {
        // Write the top of the HTML page.
        fileOut.println("<!DOCTYPE html>");
```



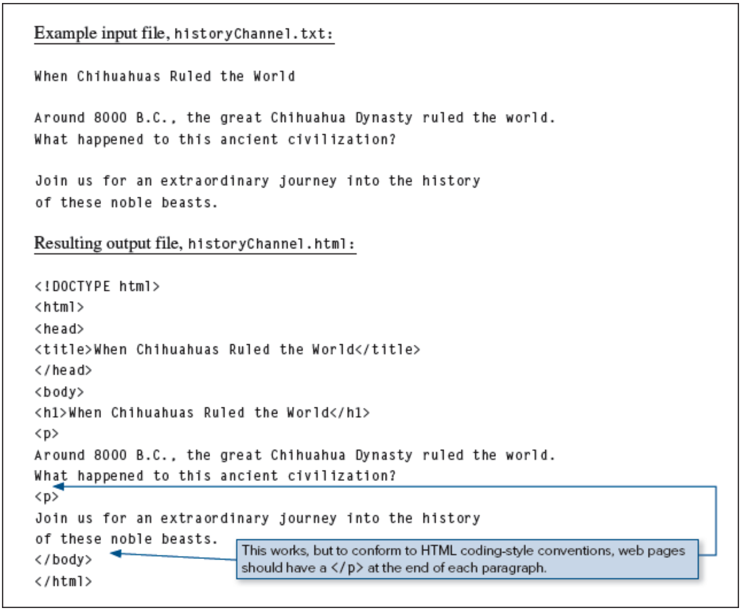
```
      else
      {
        // Write the top of the HTML page.
        fileOut.println("<!DOCTYPE html>");
        fileOut.println("<html>");
        fileOut.println("<head>");
        fileOut.println("<title>" + line + "</title>");
        fileOut.println("</head>");
        fileOut.println("<body>");
        fileOut.println("<h1>" + line + "</h1>");

        while (fileIn.hasNextLine())
        {
          line = fileIn.nextLine();

          // Blank lines generate p tags.
          if (line.isEmpty())
          {
            fileOut.println("<p>");
          }
          else
          {
            fileOut.println(line);
          }
        } // end while

        // Write ending HTML code.
        fileOut.println("</body>");
        fileOut.println("</html>");
      } // end else
    } // end try and close fileOut and fileIn automatically

    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    } // end catch
  } // end main
} // end class HTMLGenerator
```

Figure 16.1b HTMLGenerator program—part B

Now let's analyze Figure 16.1b, the second half of the HTMLGenerator program. In the `else` clause, the code writes `<html>` and `<head>` tags to the output file. It writes the input file's first line to the output file, surrounded by `<title>` and `</title>` tags.

Then it ends the web page's head section by writing `</head>` to the output file, and it begins the web page's body section by writing `<body>` to the output file. Then it reuses the input file's first line and writes it to the output file, surrounded by `<h1>` and `</h1>` tags. Then it loops through the subsequent lines in the input file. For each blank line, it writes a `<p>` tag to the output file, indicating a new paragraph. It writes each line that's not blank to the output file as is. The last two statements in the `else` clause complete the output file's HTML code. The two files close automatically when the `try` block ends. The generic `catch` block catches and describes any type of exception that might be thrown.

To see how the HTMLGenerator program works when applied to an actual input file, study the input file and resulting output file in Figure 16.2. If you'd like to verify that the HTMLGenerator program generates a working web page, create Figure 16.2's `historyChannel.txt` file, and run the HTMLGenerator program with `historyChannel.txt` as input. That should generate the `historyChannel.html` file in Figure 16.2. Open a browser window, and within that browser window, open the `historyChannel.html file`. For example, open a Windows Internet Explorer browser and perform a File/Open command. Voila—you should see the `historyChannel.html` file displayed as a web page!



Example input file, historyChannel.txt:

```
When Chihuahuas Ruled the World

Around 8000 B.C., the great Chihuahua Dynasty ruled the world.
What happened to this ancient civilization?

Join us for an extraordinary journey into the history
of these noble beasts.
```

Resulting output file, historyChannel.html:

```
<!DOCTYPE html>
<html>
<head>
<title>When Chihuahuas Ruled the World</title>
</head>
<body>
<h1>When Chihuahuas Ruled the World</h1>
<p>
Around 8000 B.C., the great Chihuahua Dynasty ruled the world.
What happened to this ancient civilization?
<p>
Join us for an extraordinary journey into the history
of these noble beasts.
</body>
</html>
```

This works, but to conform to HTML coding-style conventions, web pages should have a </p> at the end of each paragraph.

Figure 16.2 Example input file for the HTMLGenerator program and its resulting output file

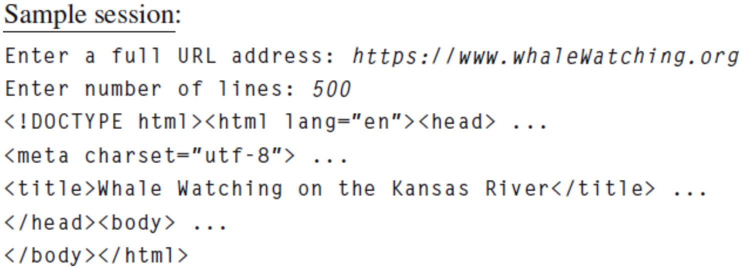
HTML tags without slashes (like <html>, <head>, and `<title>` tags) are called *start tags*. HTML tags with slashes (like `</title>`, `</head>`, and `</html>` tags) are called *end tags*. For most tags, it is required that you have an accompanying end tag for every start tag. You can see that that rule is followed for the <html>,

<head>, and `<title>` start tags in Figure 16.2, but it's not followed for the `<p>` start tags. Although not required by the HTML standards, having a `</p>` end tag for every `<p>` start tag can help with a web page's understandability and maintainability. An end-of-chapter exercise asks you to improve the HTMLGenerator program so that it generates `</p>` end tags at the end of every paragraph. Figure 16.2's callout shows where the `</p>` end tags should go in the generated `historyChannel.html` file.

16.3 A Website Reader

From accessing a file, it's a relatively small step to accessing a remote website. In either case, while data is in transit, it travels as a stream of bytes. In the previous section, when you read text from a file, the stream of bytes was hidden in the background. But when you read text from a remote website, this stream of bytes appears explicitly as an instance of the `InputStream` class.

Figure 16.3 shows a program that displays the code that forms the web page for a web page that the user specifies. The following hypothetical sample session shows what happens when the program executes and the user enters the web address for a whale watching web page.

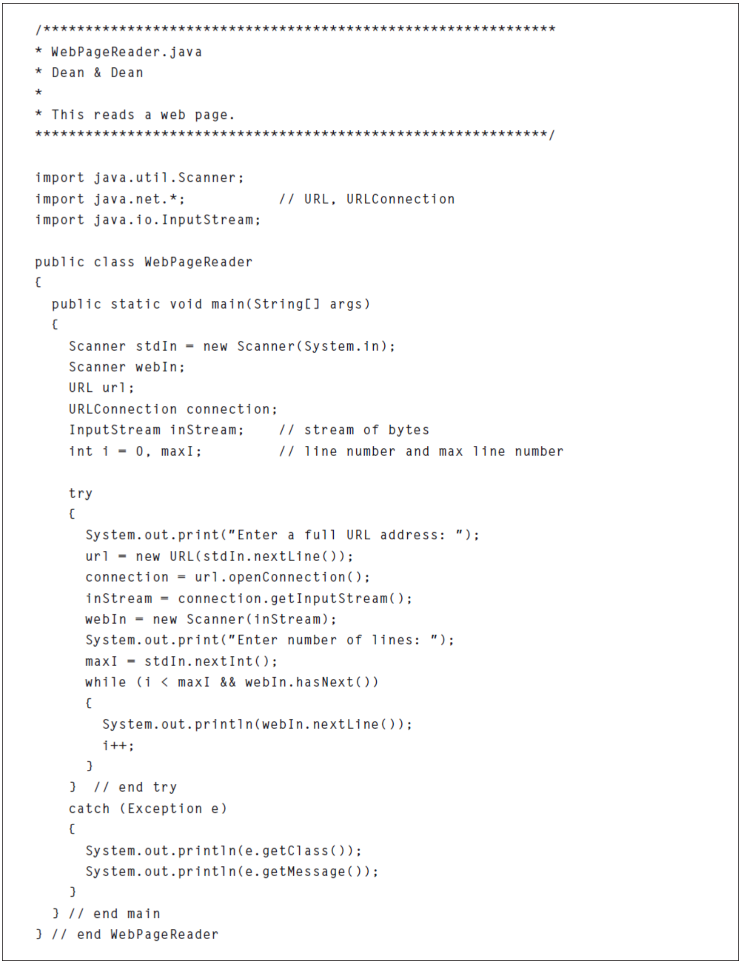


Sample session:

```
Enter a full URL address: https://www.whaleWatching.org
Enter number of lines: 500
<!DOCTYPE html><html lang="en"><head> ...
<meta charset="utf-8"> ...
<title>Whale Watching on the Kansas River</title> ...
</head><body> ...
</body></html>
```

We're assuming that the sample session's "number of lines" input of 500 was sufficient to print all of the lines of html code that form the website.

In addition to `Scanner`, the program imports packages that give access to the `URL`, `URLConnection`, and `InputStream` classes. After making declarations, the program enters a `try` block. The `try` block is required because the URL constructor can throw a `MalformedURLException`, the `openConnection` method call can throw an `IOException`, and the `getInputStream` method call can throw an `IOException`. After successfully completing its dangerous operations, the program asks for a maximum number of lines to print. Then it enters a `while` loop that prints up to this number of lines.



```
/*************************************************************
* WebPageReader.java
* Dean & Dean
*
* This reads a web page.
*************************************************************/

import java.util.Scanner;
import java.net.*;            // URL, URLConnection
import java.io.InputStream;

public class WebPageReader
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    Scanner webIn;
    URL url;
    URLConnection connection;
    InputStream inStream;    // stream of bytes
    int i = 0, maxI;         // line number and max line number

    try
    {
      System.out.print("Enter a full URL address: ");
      url = new URL(stdIn.nextLine());
      connection = url.openConnection();
      inStream = connection.getInputStream();
      webIn = new Scanner(inStream);
      System.out.print("Enter number of lines: ");
      maxI = stdIn.nextInt();
      while (i < maxI && webIn.hasNext())
      {
        System.out.println(webIn.nextLine());
        i++;
      }
    } // end try
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end WebPageReader
```

Figure 16.3 WebPageReader program—a simple website reader

16.4 Object File I/O

When file data is text, you can write to a file with a simple word processor (like Windows Notepad or UNIX vi) and read from that file with a Java program, or vice versa. When a file is not text, it's more difficult, but when the programs that write and read a particular file are both Java programs, you have an advantage. You can use software that's built into the Java language to perform the structural conversion between program objects and streams of bytes. This section explains how to use that built-in software.

Enabling Java Objects to be Stored in a File

The Java language has a built-in mechanism that *serializes* each object's data as it goes into a file and *unserializes* that data as it comes out of a file and goes back into object form. Whenever a program writes serialized data into a file, it also writes the recipe it used to serialize that data. That recipe includes the type of the object, the type of each data item, and the sequence in which the data items are stored. When another program reads serialized data from a file, it also reads the recipe to learn how to reconstruct the object from the serialized data. To enable a class to use Java's built-in serializing mechanism, you must append the following clause to that class's heading:

implements Serializable

This makes it look as though your class is implementing an interface. But this interface doesn't define any named constants, and it doesn't require the class to implement any particular methods. It just identifies the class's objects as objects needing serialization services. For example, look at the `TestObject` class in Figure 16.4. Notice that this class implements the

`Serializable` interface.

If a class is `Serializable`, all classes derived from it are automatically `Serializable`, too. Suppose your `Serializable` class has instance variables that refer to other objects. Those objects' classes also must be `Serializable`. This must be true through all levels in a composition hierarchy. Does that sound like a pain? It's not, really. Just be sure to include `implements Serializable` in the definition of all classes that define objects that you'd like to store as objects. The alternative would be a pain, though. If you couldn't store a whole object, you'd have to provide explicit code to write and read each primitive data item in the container object, and in all component objects in that container object, down the composition tree to all primitive leaves.

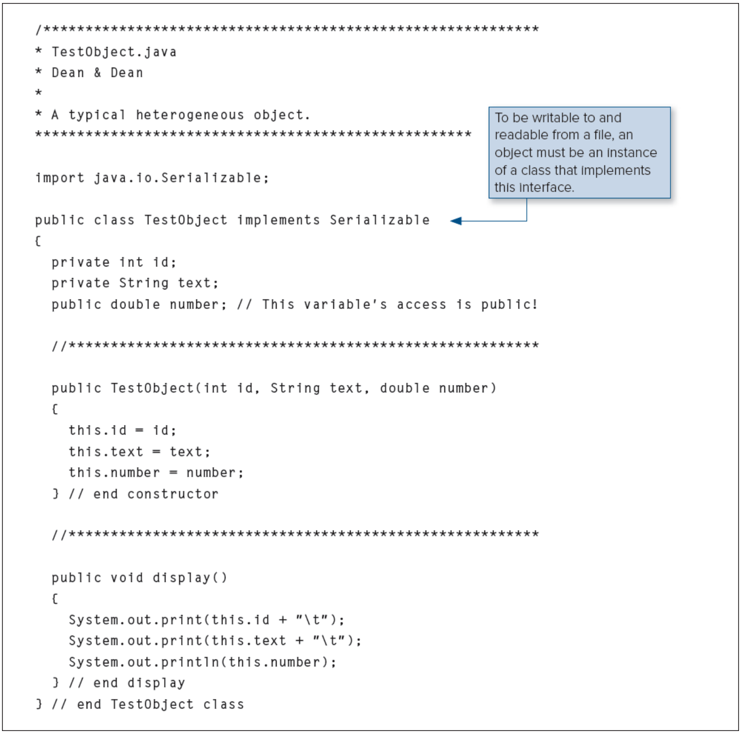
If you define or extend a class that

implements `Serializable`, to verify that the class read into is compatible with the class written from, the compiler automatically gives that class a distinctive version number. You can override this automatically generated version number with an explicit version number by including a declaration like this:

private static final long serialVersionUID = 0L;

Then, each time you modify the class, you can increment the version number—to 1L, 2L, 3L, etc. If you use the `-Xlint` compiler option to show all warnings, the compiler issues a warning if you do not give a `Serializable` class an explicit version number. For all situations we consider in this book, you can safely refrain from providing an explicit version number and safely ignore the corresponding compiler warning that might occur.

Notice that one of the instance variables in the `TestObject` class in Figure 16.4 is `public`. In general, making an instance variable `public` is not good practice, but we do it here to make it easier for you to understand certain modifications to a later program that uses this particular class.



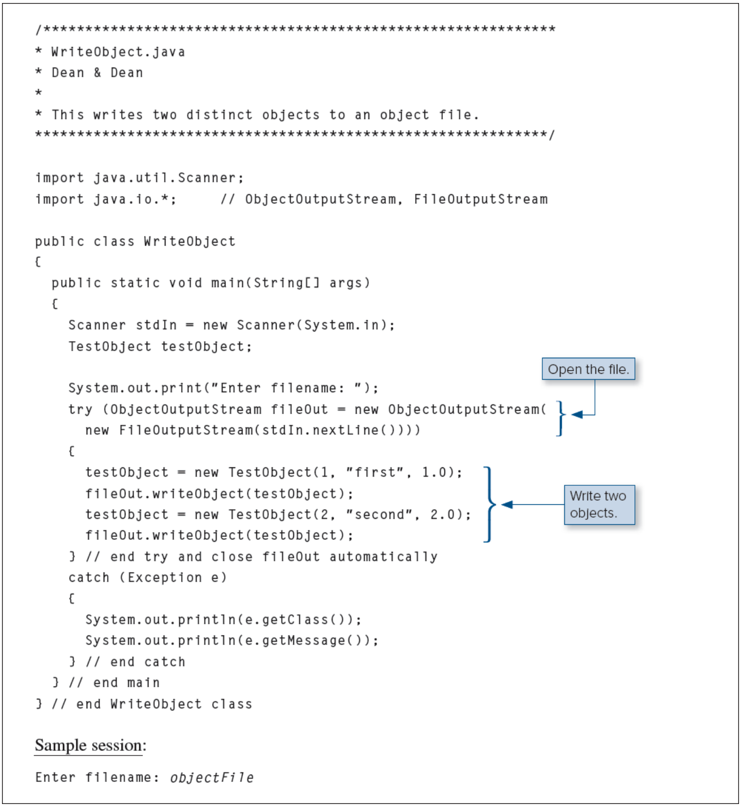
```
/**********************************************************
* TestObject.java
* Dean & Dean
*
* A typical heterogeneous object.
***************************************************

import java.io.Serializable;

public class TestObject implements Serializable
{
  private int id;
  private String text;
  public double number; // This variable's access is public!

  //*******************************************************

  public TestObject(int id, String text, double number)
  {
    this.id = id;
    this.text = text;
    this.number = number;
  } // end constructor

  //*******************************************************

  public void display()
  {
    System.out.print(this.id + "\t");
    System.out.print(this.text + "\t");
    System.out.println(this.number);
  } // end display
} // end TestObject class
```

To be writable to and readable from a file, an object must be an instance of a class that implements this interface.

Figure 16.4 Typical definition of a Serializable object

Writing a Serializable Object to a File

Figure 16.5 contains a program that writes instances of `TestObject` to a user-specified file. The try- with-resources header opens the user-specified file by instantiating a `FileOutputStream` and then an `ObjectOutputStream`. Inside the `try` block, the program instantiates and writes two distinct objects. Upon reaching the end of the `try` block, the JVM automatically closes the file. The generic `catch` block describes any exception that might be thrown.



```
/*************************************************************
* WriteObject.java
* Dean & Dean
*
* This writes two distinct objects to an object file.
*************************************************************/

import java.util.Scanner;
import java.io.*;     // ObjectOutputStream, FileOutputStream

public class WriteObject
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    TestObject testObject;

    System.out.print("Enter filename: ");
    try (ObjectOutputStream fileOut = new ObjectOutputStream(
      new FileOutputStream(stdIn.nextLine())))
    {
      testObject = new TestObject(1, "first", 1.0);
      fileOut.writeObject(testObject);
      testObject = new TestObject(2, "second", 2.0);
      fileOut.writeObject(testObject);
    } // end try and close fileOut automatically
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    } // end catch
  } // end main
} // end WriteObject class
```

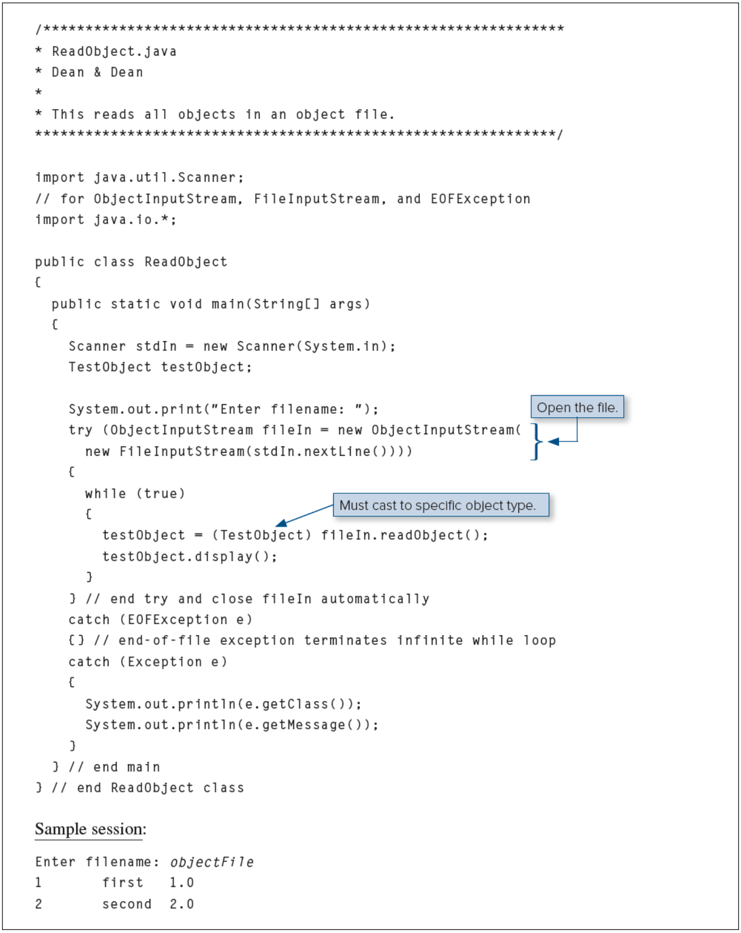
Sample session:

```
Enter filename: objectFile
```

Figure 16.5 WriteObject program, which writes Serializable objects to a file

Reading a Serializable Object from a File

Figure 16.6's ReadObject program reads data for all objects of the `TestObject` class from a user-specified file. The code down through the try-with-resources header should look familiar—because it is similar to the code down through the try-with-resources header in the preceding WriteObject program.



```
/*******************************************************************
* ReadObject.java
* Dean & Dean
*
* This reads all objects in an object file.
*******************************************************************/

import java.util.Scanner;
// for ObjectInputStream, FileInputStream, and EOFException
import java.io.*;

public class ReadObject
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    TestObject testObject;

    System.out.print("Enter filename: ");
    try (ObjectInputStream fileIn = new ObjectInputStream(
      new FileInputStream(stdIn.nextLine())))
    {
      while (true)
      {
        testObject = (TestObject) fileIn.readObject();
        testObject.display();
      }
    } // end try and close fileIn automatically
    catch (EOFException e)
    {} // end-of-file exception terminates infinite while loop
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end ReadObject class
```

Open the file.

Must cast to specific object type.

Sample session:

```
Enter filename: objectFile
1       first   1.0
2       second  2.0
```

Figure 16.6 ReadObject class that reads all Serializable objects from a file

Within its `try` block, the ReadObject program contains an infinite `while` loop, which keeps reading objects until it throws an `EOFException` (end-of-file exception). When reading each object, we must cast the returned value explicitly to our specific object type. The first `catch` block does nothing except catch the `EOFException` and terminate the infinite `while` loop. We're forced to use this odd loop-termination trick because the `ObjectInputStream` class does not have anything comparable to the `hasNext` method you'd normally expect to see in the parentheses after `while`. But this trick works just fine, because try-with-resources automatically performs the necessary file closing. The subsequent generic `catch` block describes any other exceptions that might be thrown in the event of a real problem.

The sample session cleanly displays the two previously written objects without fussing about the `EOFException`, which terminated the `while` loop.

Writing an Updated Version of a Previously Written Object

If you ask `ObjectOutputStream`'s `writeObject` method to write exactly the same object again while the file is still open, the serializing software recognizes the repetition and writes just a reference to the previously written object. This is like what happens when you instantiate a new `String` that is exactly the same as a previously instantiated `String`. This is a nice space-saving feature, but it can be a problem if you're simulating the behavior of a particular object, and you want a file to accumulate a record of that object's changing state as the simulation progresses. To see this problem, in the WriteObject program in Figure 16.5, replace this statement:

```
testObject = new TestObject(2, "second", 2.0);
```

with this:

```
testObject.number += .1;
```

Then, if you execute the revised WriteObject program and re-execute the ReadObject program, the ReadObject program will generate this output:

```
Enter name of file to read: objectFile
1       first    1.0
1       first    1.0
```

The second record of the object's state is just a copy of the first record. It doesn't reflect the change in the value of the `number` variable. To make Java store the latest state of an object instead of just a reference to the original state, you need to invoke `ObjectOutputStream`'s `reset` method sometime before you write the updated version of the object. To see how this works, in the WriteObject program in Figure 16.5, this time, replace

```
    testObject = new TestObject(2, "second", 2.0);
```

with this:

```
    fileOut.reset();
    testObject.number += .1;
```

This allows writing an updated version of a previously written object.

Then execute the revised WriteObject program and the ReadObject program, and you’ll get the result you want:

```
Enter name of file to read: objectFile
1       first   1.0
1       first   1.1
```

16.5 Character Sets and File-Access Options

This section describes ways to read and write text files with alternate character sets and ways to specify file-access options.

Character Sets

In the ReadFromFile programs in Figures 15.12a and 15.13a, the WriteToFile program in Figure 15.17, and the HTMLGenerator program in

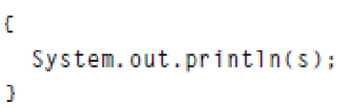
Figures 16.1a and 16.1b, all text representation employed a certain set of characters—the current computer's *default character set*. Section 16.3's sample session shows that the targeted website uses the UTF-8 character set. (UTF stands for "Unicode Transformation Format" and the "8" identifies the version that uses 8 bits to represent 256 different characters.) You can determine the name of the default character set your computer uses by importing

```
    java.nio.charset.Charset
```

and calling

```
    Charset.defaultCharset()
```

Java guarantees support for certain standard character sets, like the 128 characters in US-ASCII (see Figures A1.1a and A1.1b in Appendix 1), the 256 characters in UTF-8, and several 16-bit Unicode formats. You can read about these in Java API documentation. You can determine the names of all the particular character sets your own computer can identify, read, and write by executing this code fragment:



```
for (String s : Charset.availableCharsets().keySet())
{
  System.out.println(s);
}
```

In our ReadFromFile, WriteToFile, and HTMLGenerator programs, we could have specified one of these other character sets by using slightly different constructors. For example, to modify the ReadFromFile program to read data that was written in the particular character set having the name “US-ASCII” instead of using the one-parameter `Scanner` constructor, use this two-parameter `Scanner` constructor:

fileIn = new Scanner(Paths.get(filename), "US-ASCII");

Suppose you write “Hello, World!” to a text file with UTF-16 (a 16-bit representation) and then read it back with US-ASCII (a 7-bit representation). What would you expect to see? Here's what we get:

```
?? H e l l o , W o r l d !
```

When we write with US-ASCII and read with UTF-16, we get this:

```
????????
```

When we write and read with consistent character sets, either both with US-ASCII or both with UTF-16, we get what you would expect:

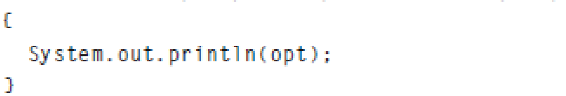
```
Hello, World!
```

As you might expect and these results suggest, a UTF-16 file is approximately twice as big as a US-ASCII file with the same number of characters.

File-Access Options

Up until now, the types of file-access options we have employed have been the default options built into the particular file-handling constructors we used. For example, whenever we opened a file for writing, the computer assumed the file did not exist and should be created from scratch. Or if the file did exist, the computer assumed we wanted to discard all its previous data and treat it as a brand new file. But sometimes we would rather add or append new data to data already in an existing file. To do this, we open the file with an `APPEND` option. This is one of the constants in `Enum StandardOpenOption`, found in the `java.nio.file` package.

You can display the names of all of Java's standard open options by executing this code:



```
for (StandardOpenOption opt : StandardOpenOption.values())
{
  System.out.println(opt);
}
```

Upcoming examples will include `StandardOpenOption`s identified by these `String` designations: APPEND, CREATE, READ, TRUNCATE_EXISTING, and WRITE.

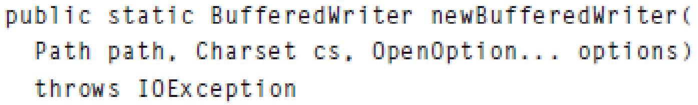
16.6 Buffered Text File I/O

The previous examples of text file reading in the ReadFromFile programs in Figures 15.12a and 15.13a, of text file writing in the WriteToFile program in Figure 15.17, and of text file I/O in Figures 16.1a and 16.1b were fine for small and local text files. But the file-handling techniques used in those examples are not suitable for large files or files that might be transferred from one computer to another.

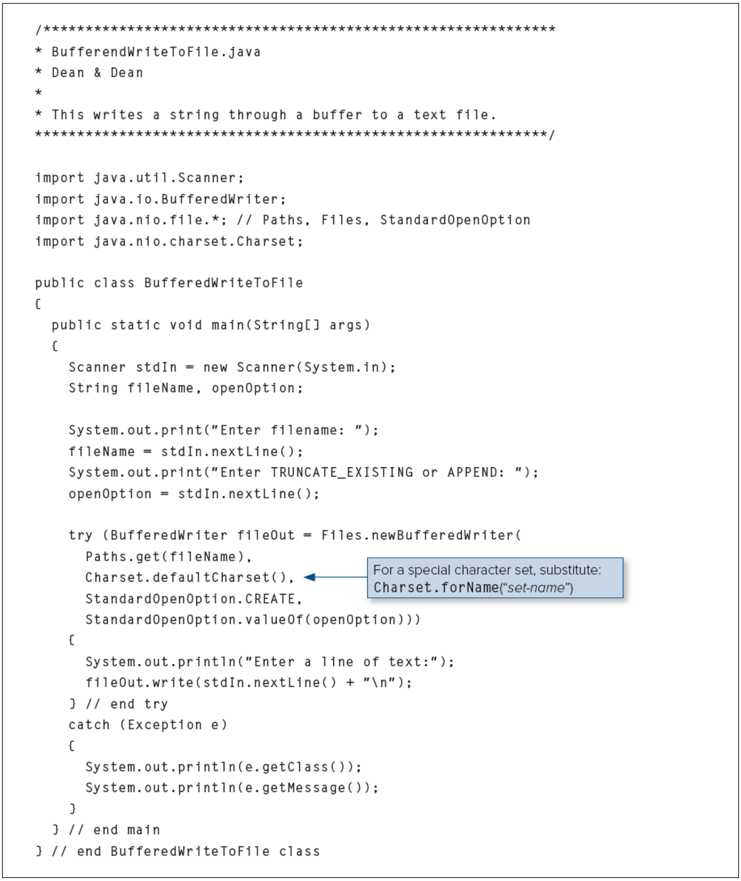
For large and/or remote file I/O, you should always employ an intermediate buffer. A *buffer* is a sequential storage structure that acts like a first-in first-out (FIFO) queue, or “waiting line.” As time passes, a queue’s length changes dynamically with independent variations in arrival rate and service rate. Because a buffer resides in high-speed memory, a program can transfer data into or out of a buffer very quickly. On the other hand, files reside in persistent storage, which is always slower than the computer's main memory, and it is occasionally located a considerable distance from the computer itself—perhaps even on the other side of the world. The computer accesses files in intermittent bursts whose timing depends on extraneous factors like disk position, other traffic on the bus or communication link, and distance.[1] A buffer decouples program transfers from file transfers, so that transfers between the program and the buffer do not need to be synchronized with transfers between the buffer and the file. With large files, a buffer provides a substantial performance advantage.

Using Java’s BufferedWriter

The BufferedWriteToFile program in Figure 16.7a shows how to write text to a file through a buffer. It also illustrates specification of character set and open options. The `java.io` package includes the BufferedWriter class. the `java.nio.file` package includes the Paths class, whose static get method returns a Path object. That package also contains the Files class, whose newBufferedWriter method returns a BufferedWriter object configured for a particular character set and particular open options. Java API documentation says the `newBufferedWriter` method conforms to this interface:



```
public static BufferedWriter newBufferedWriter(
  Path path, Charset cs, OpenOption... options)
  throws IOException
```



```
/*************************************************************
* BufferendWriteToFile.java
* Dean & Dean
*
* This writes a string through a buffer to a text file.
*************************************************************/

import java.util.Scanner;
import java.io.BufferedWriter;
import java.nio.file.*; // Paths, Files, StandardOpenOption
import java.nio.charset.Charset;

public class BufferedWriteToFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName, openOption;

    System.out.print("Enter filename: ");
    fileName = stdIn.nextLine();
    System.out.print("Enter TRUNCATE_EXISTING or APPEND: ");
    openOption = stdIn.nextLine();

    try (BufferedWriter fileOut = Files.newBufferedWriter(
      Paths.get(fileName),
      Charset.defaultCharset(),
      StandardOpenOption.CREATE,
      StandardOpenOption.valueOf(openOption)))
    {
      System.out.println("Enter a line of text:");
      fileOut.write(stdIn.nextLine() + "\n");
    } // end try
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end BufferedWriteToFile class
```

For a special character set, substitute: `Charset.forName`("*set-name*")

Figure 16.7a BufferedWriteToFile class This demonstrates Files.newBufferedWriter with default character set and standard open options. The callout shows entry for a special character set.

The character-set and open-options

parameters do not appear in the BufferedWriter constructors in the java.io package, and we'd need lots of extra code to include them if the Files.newBufferedWriter method were not available.

The OpenOption... notation is called a *varargs*. It means that we may supply any number of arguments of the specified type, including none. A varargs works for any type of object, but in this particular case, the specified type is OpenOption. Only the last parameter may be a varargs parameter. The argument passed to a varargs parameter may be either an array or a sequence of objects separated by commas. In the Java API, varargs are frequently enums, and sure enough, classes that implement the OpenOption interface, like StandardOpenOption, are enums. (For a discussion of enum, see Section 12.13.)

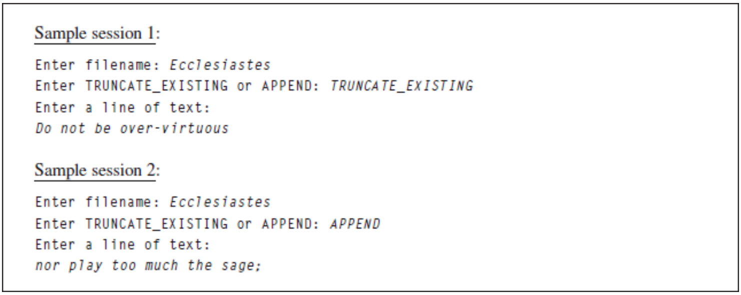
After prompting for and reading a

filename, the program prompts for and reads a String designating the second of two open options. In a try-with-resources header, it asks Files to return an appropriately configured instance of BufferedWriter. Our BufferedWriteToFile class uses the local computer's default character set, but the callout indicates that we could specify another character set with something like Charset.forName("US-ASCII"). StandardOpenOption's valueOf method converts the String designation for the second open option into its internal value.

Within the `try` block, the program asks the user to enter a line of text. Then it writes that line to the file. The explicit "\n" in the write method's argument restores the line termination that `Scanner`'s `nextLine` method automatically removes. For simplicity, this program reads and writes just one line, but it would be straightforward to include a loop that reads and writes any number of lines.

The `BufferedWriter` instance closes automatically at the end of the `try` block. We use a generic `catch` block because the newBufferedWriter method call can throw an `InvalidPathException`. This is an unchecked exception, which does not descend from the checked `IOException` we must catch. Our generic display is easier to read than the default stack trace generated by the JVM if the path is invalid.

Figure 16.7b shows two sample-session outputs. In the first sample session, the user specifies the second open option as "TRUNCATE_EXISTING" to delete all existing text from any existing file whose name happens to match the specified filename. Then the user enters what will be the first line of text in the specified file. In the second sample session, the user specifies the second open option as “APPEND” to append the subsequently entered line of text.



Sample session 1:

```
Enter filename: Ecclesiastes
Enter TRUNCATE_EXISTING or APPEND: TRUNCATE_EXISTING
Enter a line of text:
Do not be over-virtuous
```

Sample session 2:

```
Enter filename: Ecclesiastes
Enter TRUNCATE_EXISTING or APPEND: APPEND
Enter a line of text:
nor play too much the sage;
```

Figure 16.7b Outputs for two executions of the BufferedWriteToFile program in Figure 16.7a APPEND and TRUNCATE_EXISTING identify particular StandardOpenOptions.

Using Java’s BufferedReader

The BufferedReadFromFile program in Figure 16.8 shows how to read text from a file through a buffer. Files’ newBufferedReader method also includes specification of the character set. It does not accept any open-option specification—it just assumes that we want the READ option only. The imports and local variables are the same as those in the BufferedWriteToFile program in Figure 16.7a. The user interaction is like that in the BufferedWriteToFile program, except it's simpler because there is no open-option specification. In the try-with-resources header, the creation of a BufferedReader is like the creation of a BufferedWriter was in Figure 16.7a, except it's simpler because there is no open-option specification.

Within the try block, the program employs a while loop to print all lines in the specified file. BufferedReader's ready method provides the same loop condition as Scanner's hasNextLine method. The BufferedReader instance closes automatically at the end of the try block. As indicated earlier, we use a generic `catch` block because the newBufferedReader method call throws an unchecked InvalidPathException, which does not descend from the checked IOException, and our generic display is easier to read than the default stack trace generated by the JVM.

The output at the bottom of Figure 16.8 displays the text written to the file in the two sample sessions in Figure 16.7b.

Web Reading

You can also use Java's BufferedReader to read from the web. To do this, create a URL object. Then create a URLConnection. Then create an InputStream. Then use that InputStream as the argument in an InputStreamReader constructor, and finally, use that InputStreamReader as the argument in a BufferedReader constructor. Here is the sequence of operations:

```
URL url = new URL(webAddress);
URLConnection connection = url.openConnection();
InputStream in = connection.getInputStream();
BufferedReader reader =
  new BufferedReader(new InputStreamReader(in));
```

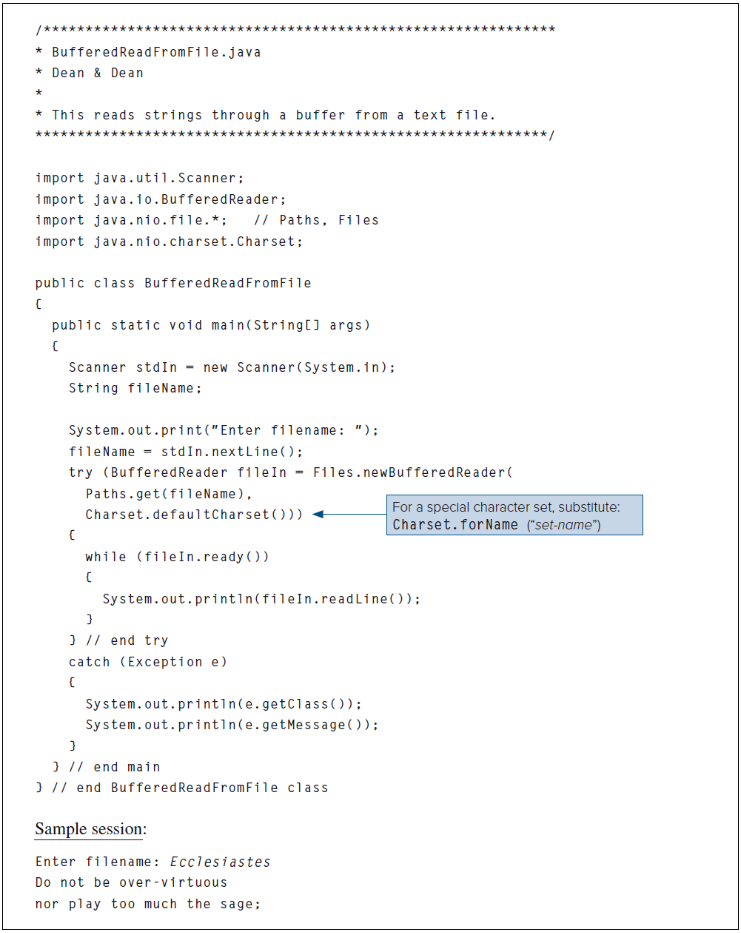
The URL and URLConnection are in the java.net package. The InputStream, InputStreamReader, and BufferedReader classes are in the java.io package.

If you need a character set different from your computer's default character set, you can specify the character set with a second argument supplied to an alternate InputStreamReader constructor. An exercise at the end of this chapter asks you to read a web page with a BufferedReader.

16.7 Primitive Buffers with Random Access

We can also buffer primitive data, like byte, char, short, int, float, long, and double. To facilitate this, the java.nio package provides the base class, Buffer, which descends directly from Object, plus Buffer's direct descendants: ByteBuffer, CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer.

These classes make it easy to read and write each kind of primitive variable. Because computer systems universally transmit and store data as bytes, the ByteBuffer class plays a central role. It provides single-variable put and get methods that make conversions between other primitive types and the byte type. The ByteBuffer class also provides methods like asIntBuffer and asDoubleBuffer, which provide alternate access to chunks of a ByteBuffer's data. With help from these methods, the data in the array underlying a ByteBuffer can represent any primitive type. Beyond that, it can represent any combination of primitive types. Still more, it can represent any combination of primitive types and Object types. We won’t try to mix primitives and objects from arbitrary classes, but a later example will put an int, a String, and a double into the same ByteBuffer.



```
/*******************************************************************
* BufferedReadFromFile.java
* Dean & Dean
*
* This reads strings through a buffer from a text file.
*******************************************************************/

import java.util.Scanner;
import java.io.BufferedReader;
import java.nio.file.*;   // Paths, Files
import java.nio.charset.Charset;

public class BufferedReadFromFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName;

    System.out.print(”Enter filename: ”);
    fileName = stdIn.nextLine();
    try (BufferedReader fileIn = Files.newBufferedReader(
      Paths.get(fileName),
      Charset.defaultCharset()))
    {
      while (fileIn.ready())
      {
        System.out.println(fileIn.readLine());
      }
    } // end try
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end BufferedReadFromFile class
```

For a special character set, substitute: `Charset.forName` (“*set-name*”)

Sample session:

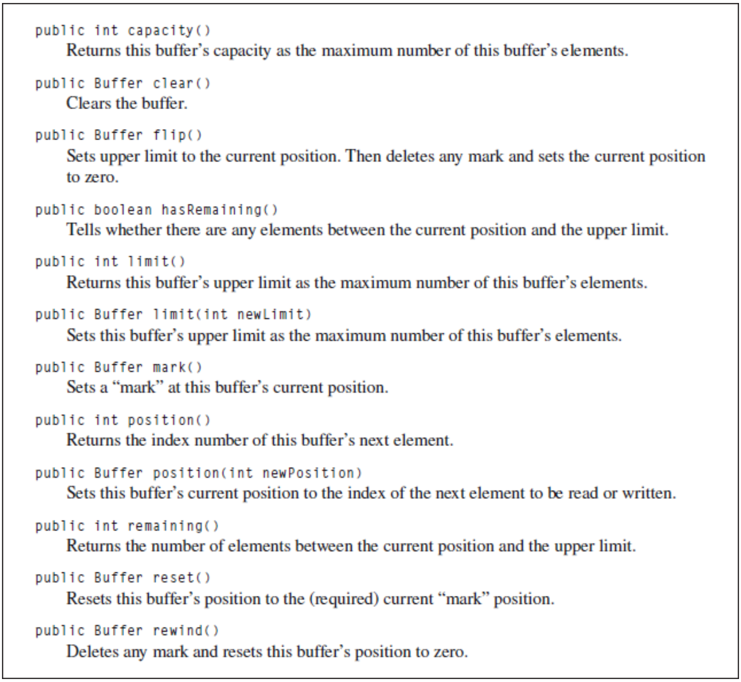
```
Enter filename: Ecclesiastes
Do not be over-virtuous
nor play too much the sage;
```

Figure 16.8 BufferedReadFromFile class,

which demonstrates BufferedReader with a default character set The callout shows an entry for a special character set.

Buffer Methods—Inherited by All Buffer Descendants

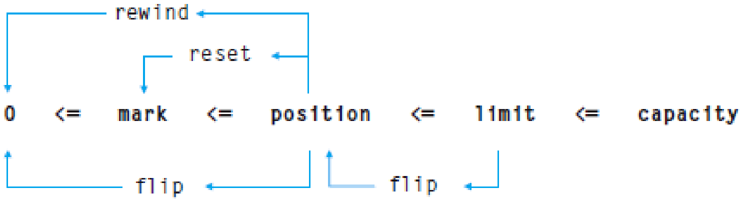
All particular primitive buffer types inherit common methods from the abstract parent class, Buffer. Figure 16.9 presents several of these Buffer methods as interfaces and short descriptions. Many of these methods would normally be void, but they return a Buffer instead. Because these returned Buffers are this references to the calling object, these Buffer methods support method-call chaining.



`public int capacity()`	Returns this buffer's capacity as the maximum number of this buffer's elements.
`public Buffer clear()`	Clears the buffer.
`public Buffer flip()`	Sets upper limit to the current position. Then deletes any mark and sets the current position to zero.
`public boolean hasRemaining()`	Tells whether there are any elements between the current position and the upper limit.
`public int limit()`	Returns this buffer's upper limit as the maximum number of this buffer's elements.
`public Buffer limit(int newLimit)`	Sets this buffer's upper limit as the maximum number of this buffer's elements.
`public Buffer mark()`	Sets a "mark" at this buffer's current position.
`public int position()`	Returns the index number of this buffer's next element.
`public Buffer position(int newPosition)`	Sets this buffer's current position to the index of the next element to be read or written.
`public int remaining()`	Returns the number of elements between the current position and the upper limit.
`public Buffer reset()`	Resets this buffer's position to the (required) current "mark" position.
`public Buffer rewind()`	Deletes any mark and resets this buffer's position to zero.

Figure 16.9 Several Buffer methods—those that return Buffer can be chained

The capacity, limit, position, and mark methods get and set corresponding buffer indices. These are like bookmarks that identify particular buffer locations where you may want to go or to which you may want to return. They are in units of the current primitive type, like the length of or index into an array of that type. They satisfy these constraints:



By default, there is no mark, the initial position is zero, and the initial limit equals capacity.

As a program reads or writes elements, position automatically increments to the start of the next element. A mark method call makes the current position be the destination of any subsequent reset. A rewind method call deletes any mark and makes the next position zero. A flip method call sets the limit to the current position, deletes any mark, and makes the next position zero.

As we have said, a buffer is like a queue or waiting line. Assume that a quantity of elements does not exceed the buffer's capacity, and suppose you want the buffer to transport that quantity to or from a file or through the Internet. In a loop, you could put these elements into the buffer. Then, in another loop, you could get these elements from the buffer and pass them on. To facilitate this operation, insert a flip method call after the put loop, and execute the get loop while(position < limit).

Your ability to set position to any index between zero and limit gives you *random read and write access* to any element(s) in the range between position(0) and position(limit-1).

Elementary ByteBuffer Methods

The ByteBuffer class defines many additional methods. The static method

public static ByteBuffer allocate(int capacity)

acts like a ByteBuffer constructor. It creates and returns an empty ByteBuffer whose maximum number of bytes equals the capacity parameter. The initial limit also equals this capacity. Any method call that tries to put into the buffer more bytes than limit throws a BufferOverflowException.

There are get methods for all primitive variables: get for byte, getChar for char, getDouble for double, getFloat for float, getInt for int, getLong for long, and getShort for short. There are also put methods for all primitive variables: put for byte, putChar for char, putDouble for double, putFloat for float, putInt for int, putLong for long, and putShort for short.

These methods are overloaded. Zero-parameter get methods like getChar() are *relative*. They read and return the primitive at the calling buffer's current position and then increment that position to the start of the next element. The position increment is two bytes for getChar, four bytes for getInt, etc. One-parameter get methods like getChar(int index) are *absolute*. The index parameter is a primitive index. It specifies the location of the desired primitive not in bytes, but in number of primitives before the desired primitive. These absolute get methods do not change the buffer's current position.

One-parameter put methods like putChar(char value) are relative. They write the primitive, starting at the calling buffer's current position. Then they increase that position to just after the inserted primitive. Two-parameter put methods like putChar(int index, char value) are absolute. The index parameter tells where the value parameter goes. These absolute put methods do not change the buffer's current position.

The relative method

public ByteBuffer put(ByteBuffer

source)

copies all remaining source bytes to the calling buffer, starting at the calling buffer's current position. As indicated in Figure 16.9, *remaining* means all bytes between current position (inclusive) and current limit (not inclusive). If the calling buffer is not big enough, the method throws a BufferOverflow Exception. The next program will use this method just to copy all data from one buffer to another. But because the two buffers' limit and position values may be different, you can use this method to copy an arbitrary subset of the data in the first buffer to an arbitrary position in the second buffer.

Figure 16.10 contains a simple program that uses ByteBuffer's relative putInt method to put a single int value at the beginning of a ByteBuffer. Then it uses ByteBuffer's absolute putDouble method to put a single double value into that same ByteBuffer, with seven blank spaces between the end of the four-byte int and the start of the eight-byte double. The first two lines of output show how the relative putInt method call adds four to buffer1's position and how the absolute putDouble method call does not change that position. Even though a double needs eight bytes, this program shows that it can start anywhere in the buffer, not just at a multiple of eight.

After populating buffer1, the program copies all its data to buffer2, using the buffer-to-buffer put method. But before doing that, it must call buffer1's rewind method. Here's why: The sample session shows that after it finishes populating buffer1, buffer1's position is 4. If the program did not have a buffer1.rewind() statement before the buffer2.put(buffer1) statement, buffer2 would not receive buffer1's first four bytes. The buffer1.rewind() statement resets buffer1's position to zero, and this enables buffer2 to receive all buffer1's data. After the buffer-to-buffer put, buffer2's position is 19, which is its limit. The buffer2.flip() statement moves that position back to zero again, preparing it for subsequent get method calls.

Array Methods

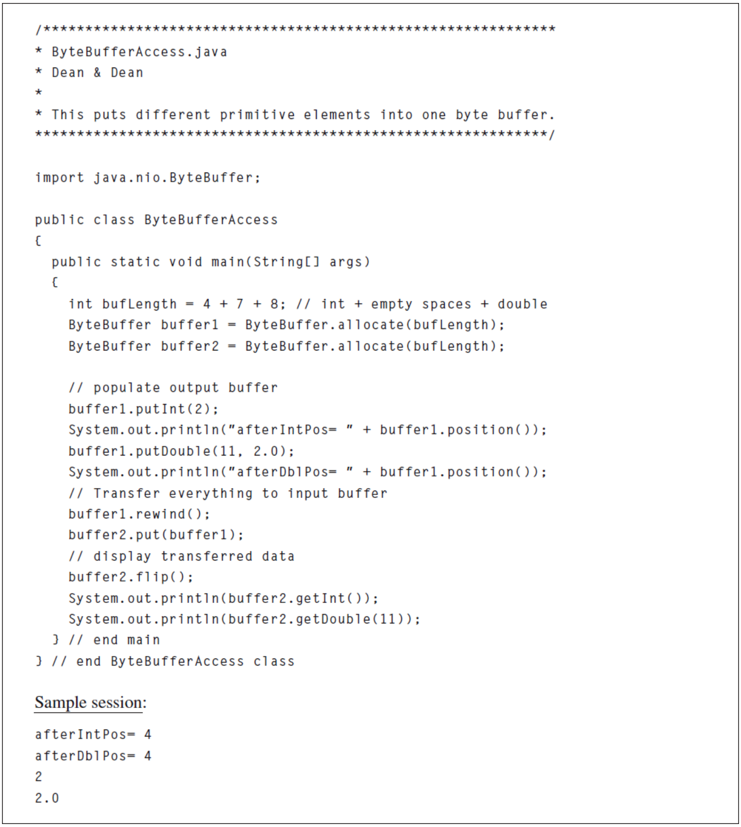
We can also copy arrays of primitives into or out of a ByteBuffer. The static method

public static ByteBuffer wrap(byte[] byteArray)

acts like another ByteBuffer constructor. It creates, populates, and returns a ByteBuffer filled with the parameter's elements. Alternatively, you can use the previously described allocate method to create the buffer and then populate it with separate method call(s). This alternative requires more code, but it is more versatile.

To understand how a ByteBuffer can help organize arrays, you need to become familiar with some additional ByteBuffer methods. The ByteBuffer class has additional overloaded get and put methods for arrays of bytes. The one-parameter get(byte[] destination) method is *relative*. Starting at the calling buffer's current position, this method copies all remaining buffer bytes into a previously defined destination array, and it increases the calling buffer's position to its limit. The three-parameter get(byte[] destination, int offset, int length) is *absolute*. It copies length bytes, starting at the buffer's current position, into the destination array, starting at offset array index. This absolute get method does not change the buffer’s current position.

The one-parameter put(byte[] source) is relative. This method copies all bytes in the source array into the calling buffer, starting at the calling buffer’s current position. It increases the calling buffer’s position by the total number of bytes written. The three-parameter put(byte[] source, int offset, int length) is absolute. It copies length bytes from the source array, starting at the offset array index, into the calling buffer, starting at the buffer’s current position. This absolute put method does not change the buffer’s current position.



```
/*************************************************************
* ByteBufferAccess.java
* Dean & Dean
*
* This puts different primitive elements into one byte buffer.
*************************************************************/

import java.nio.ByteBuffer;

public class ByteBufferAccess
{
  public static void main(String[] args)
  {
    int bufLength = 4 + 7 + 8; // int + empty spaces + double
    ByteBuffer buffer1 = ByteBuffer.allocate(bufLength);
    ByteBuffer buffer2 = ByteBuffer.allocate(bufLength);

    // populate output buffer
    buffer1.putInt(2);
    System.out.println("afterIntPos= " + buffer1.position());
    buffer1.putDouble(11, 2.0);
    System.out.println("afterDblPos= " + buffer1.position());
    // Transfer everything to input buffer
    buffer1.rewind();
    buffer2.put(buffer1);
    // display transferred data
    buffer2.flip();
    System.out.println(buffer2.getInt());
    System.out.println(buffer2.getDouble(11));
  } // end main
} // end ByteBufferAccess class
```

Sample session:

```
afterIntPos= 4
afterDblPos= 4
2
2.0
```

Figure 16.10 ByteBufferAccess program has different types of elements in the same buffer

Another ByteBuffer method, public byte[] array(), returns a byte array that is a *view* or an alias of the array that underlies the calling buffer. Changes to the returned array appear immediately as changes in the calling buffer, and vice versa.

The ByteBuffer class provides the additional view methods asCharBuffer, asDoubleBuffer, asFloatBuffer, asIntBuffer, asLongBuffer, and asShortBuffer. These additional view methods give other types of primitives access to a ByteBuffer's underlying array. The returned primitive-buffer view starts at the ByteBuffer's current position. This returned view must be assigned to a declared variable, like DoubleBuffer doubleView, but it is not a separate object. Therefore, it doesn't need to be created with a separate allocate.

Other classes in the java.nio package—CharBuffer, DoubleBuffer, FloatBuffer, IntBuffer, LongBuffer, and ShortBuffer—define get methods like get(char[] destination) and get(char[] destination, int offset, int length). Like ByteBuffer's corresponding get methods, these copy a range of primitives from the calling buffer to a previously defined destination array. These other classes (CharBuffer, etc.) also define put methods like put(char[] source) and put(char[] source, int offset, int length). Like ByteBuffer's corresponding put methods, these methods copy source array values into the calling buffer.

With these methods, it's relatively easy to copy an array of primitives that are not bytes to or from a ByteBuffer. For example, given double[] doubles, and ByteBuffer buffer, we can copy the doubles array into the byte buffer with a statement like this:

```
buffer.asDoubleBuffer().put(doubles)
```

Or we can copy double values from the byte buffer into the doubles array with a statement like this:

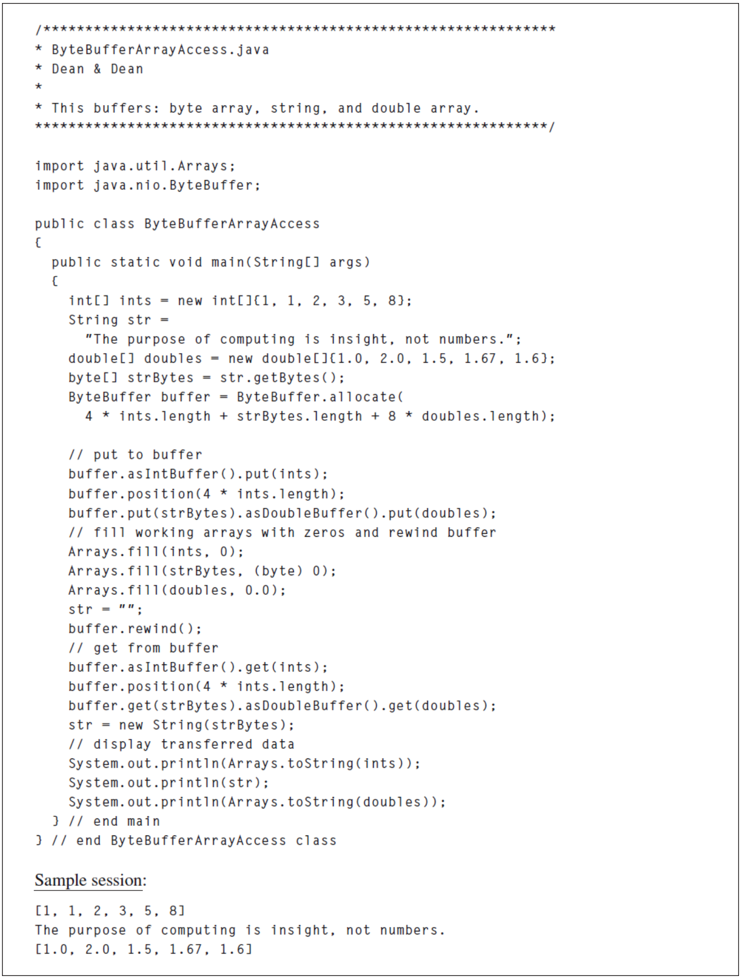
```
buffer.asDoubleBuffer().get(doubles)
```

A powerful feature of the combination of a ByteBuffer and the DoubleBuffer obtained from ByteBuffer's asDoubleBuffer method is the independence of the two buffers' position variables. Whenever you use the DoubleBuffer's put or get, it may or may not (depending on the DoubleBuffer method) change that DoubleBuffer's position. But it never changes the corresponding ByteBuffer's position, and vice versa. The zero position in the ByteBuffer is the initial byte in the underlying array. But the zero position in the DoubleBuffer is the current position in the ByteBuffer. This refers to the place in the underlying array where eight-byte double values begin. Because the two buffer positions are independent, you can change ByteBuffer's position without affecting Double Buffer's position, and vice versa. Each unit change in DoubleBuffer's position corresponds to eight bytes in the underlying array and therefore a change of eight in

ByteBuffer’s position.

The ByteBufferArrayAccess program in Figure 16.11 shows how to put and get arrays of primitive elements—an array of ints and an array of doubles. It also shows how to put and get a String. To put a String into a ByteBuffer, we first convert it to an array of bytes, and then we put that array of bytes into the ByteBuffer just as we would put an array of primitive bytes into it. To get a String from a ByteBuffer, we first get into a byte array the part of the ByteBuffer that represents the String, and then we pass that byte array to a String constructor.

The first three declarations specify the data that will go into the buffer—an array of ints, a String,[2] and an array of doubles. The next declaration converts the String to an array of bytes. The last declaration creates the buffer with just enough capacity to hold the int array, the String, and the double array. By the way, when working with buffers, it’s important to have independent knowledge of how big the whole buffer needs to be, the type(s) of data it will hold, and if more than one type, the amounts of each type. Without this independent information, you don’t know how much capacity to allocate, you don’t know what methods to use to put and get the data, and you don’t know where each type of data starts.



```
/*******************************************************************
* ByteBufferArrayAccess.java
* Dean & Dean
*
* This buffers: byte array, string, and double array.
*******************************************************************/

import java.util.Arrays;
import java.nio.ByteBuffer;

public class ByteBufferArrayAccess
{
  public static void main(String[] args)
  {
    int[] ints = new int[]{1, 1, 2, 3, 5, 8};
    String str =
      "The purpose of computing is insight, not numbers.";
    double[] doubles = new double[]{1.0, 2.0, 1.5, 1.67, 1.6};
    byte[] strBytes = str.getBytes();
    ByteBuffer buffer = ByteBuffer.allocate(
      4 * ints.length + strBytes.length + 8 * doubles.length);

    // put to buffer
    buffer.asIntBuffer().put(ints);
    buffer.position(4 * ints.length);
    buffer.put(strBytes).asDoubleBuffer().put(doubles);
    // fill working arrays with zeros and rewind buffer
    Arrays.fill(ints, 0);
    Arrays.fill(strBytes, (byte) 0);
    Arrays.fill(doubles, 0.0);
    str = "";
    buffer.rewind();
    // get from buffer
    buffer.asIntBuffer().get(ints);
    buffer.position(4 * ints.length);
    buffer.get(strBytes).asDoubleBuffer().get(doubles);
    str = new String(strBytes);
    // display transferred data
    System.out.println(Arrays.toString(ints));
    System.out.println(str);
    System.out.println(Arrays.toString(doubles));
  } // end main
} // end ByteBufferArrayAccess class
```

Sample session:

```
[1, 1, 2, 3, 5, 8]
The purpose of computing is insight, not numbers.
[1.0, 2.0, 1.5, 1.67, 1.6]
```

Figure 16.11 ByteBufferArrayAccess program has different types of arrays in the same buffer

After creating local variables, the program populates the buffer. First, buffer obtains an IntBuffer view. Then, in a chained method call, it uses that returned IntBuffer view to put the array of ints into the buffer. Because putting through the IntBuffer view does not alter buffer’s position, we need the subsequent explicit position statement to advance buffer’s position to where the next data goes. The next statement puts the strbytes array into the buffer. Then, in a chained method call, buffer obtains a DoubleBuffer view that starts just after the last byte of strbytes. Then, in another chained method call, the returned DoubleBuffer view puts the array of doubles into the buffer.

The asIntBuffer and asDoubleBuffer method calls return objects of the classes, IntBuffer and DoubleBuffer, respectively. So the methods that these returned objects call in chained operations are IntBuffer and DoubleBuffer methods. But it is not necessary to import the IntBuffer and

DoubleBuffer classes because these class names never appear explicitly in the program.

Next, the program employs some static methods from the Java API Arrays class in the java.util package to zero out all the previously filled working arrays and the String variable, str. This guarantees that our final results are not just leftover initial values. Then buffer1 rewinds itself to prepare for subsequent get operations.

The statements that get data from the buffer are similar to the statements that put data into the buffer, except they use get methods rather than put methods. The String constructor reverses the getBytes method call in the declarations.

The last three statements display the retrieved data, with help from toString methods defined in Java's Arrays class. The output confirms that the values retrieved from the buffer are exactly the same as the values previously put into the buffer.

Character Sets Revisited

For simplicity, the getBytes method call that converts the specified String to an array of bytes in the declarations, and the later String constructor with the byte[] argument both use the default character set. To work with a different character set, we could use the getBytes method and the String constructor that accept character-set specification. Specifically, we could include this additional import:

```
import java.nio.charset.Charset;
```

Then, to convert the specified String to a byte array, we could replace the strBytes declaration with this:

```
byte[] strBytes = str.getBytes(Charset.forName("US-ASCII"));
```

And later, to convert the byte array back into a String, we could use this:

```
str = new String(strBytes,
Charset.forName("US-ASCII"));
```

Of course, the choice of “US-ASCII” here is arbitrary. In practice, if the default character set is not OK, you might use one of the other character sets.

16.8 Channel I/O and Memory-Mapped Files

ByteBuffer methods organize in sequences of primitives or arrays of primitives. FileChannel methods organize in sequences of buffers or arrays of buffers. A channel is a large-scale view of what’s in a file. To put heterogeneous data into a file, first put primitives into buffers, and then put those buffers into channels.

Conversely, to get heterogeneous data from a file, first extract the buffers from the channels, and then extract the primitives from the buffers.

Channel position and size are measured in bytes just like ByteBuffer position and limit. If you don’t alter the channel’s position explicitly, position automatically increments as you write bytes. Like an ArrayList a channel’s size expands automatically to accept more data. If you specify a starting place, you could go back and overwrite any sequence of bytes. Then you could use channel’s size method to restore its position to just after the last contained byte and continue from there. After writing from any combination of buffers, you can read with a different combination of buffers. But it’s not like object I/O. To retrieve information properly, you must remember exactly where everything is and exactly what type it is.

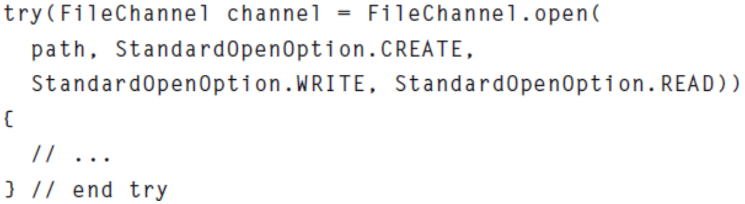
To work with file channels, include these imports:

```
import java.nio.channels.FileChannel;
import java.nio.file.*; // Path, Paths, Files, StandardOpenOption
```

To specify the file you want, create a new Path, using something like this:

```
System.out.print("Enter filename: ");
Path path = Paths.get(stdIn.nextLine());
```

To open the file for either reading or writing (over any pre-existing data), in a try-with-resources header, create a channel like this:



```
try(FileChannel channel = FileChannel.open(
  path, StandardOpenOption.CREATE,
  StandardOpenOption.WRITE, StandardOpenOption.READ))
{
  // ...
} // end try
```

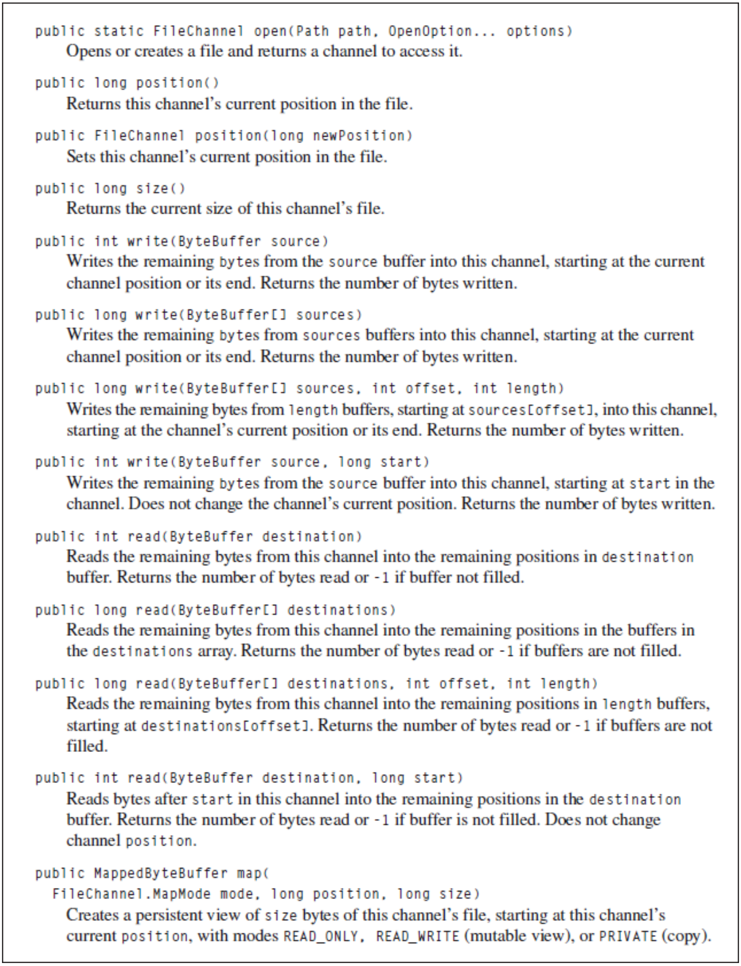
Because the file opens in a try-with resources header, it closes automatically at the end of the try block.

FileChannel Methods

To write data to a file through a channel, first write the data to one or more buffers with ByteBuffer write methods, as described in the previous section. Then use one of FileChannel's four write methods in Figure 16.12 to copy the data from the buffer(s) to the channel, and thus to the file. The channel expands automatically as required to accept whatever is written to it. The first three write methods in Figure 16.12 start writing at the channel's current position, or at the end of the channel if the channel-opening statement included StandardOpenOption.APPEND. With any of these three methods, channel position automatically increments as new data arrives.

The first write method copies data from a single source buffer, starting at that buffer’s current position. The second write method copies data from a sequence of source buffers, starting at each one’s current position. The third write method copies data from only those buffers in a specified subarray of an array of buffers. The fourth write method is like the first write method, except it starts writing at a user-specified byte index in the channel, and the channel’s internal position does not change.

To read data from a file through a channel, use one of FileChannel’s four read methods in Figure 16.12. All four of these read methods keep reading from the channel until the remaining spaces in the target buffers are filled. The remaining spaces are those from the current position to the limit. For homogeneous buffers, size each buffer to match a coherent range of bytes in the channel, so that each buffer’s capacity determines the number of bytes copied. For a heterogeneous buffer, set position and limit so that the difference between them determines the number of bytes copied.



`public static FileChannel open(Path path, OpenOption... options)`
Opens or creates a file and returns a channel to access it.

`public long position()`
Returns this channel's current position in the file.

`public FileChannel position(long newPosition)`
Sets this channel's current position in the file.

`public long size()`
Returns the current size of this channel's file.

`public int write(ByteBuffer source)`
Writes the remaining `bytes` from the `source` buffer into this channel, starting at the current channel position or its end. Returns the number of bytes written.

`public long write(ByteBuffer[] sources)`
Writes the remaining `bytes` from `sources` buffers into this channel, starting at the current channel position or its end. Returns the number of bytes written.

`public long write(ByteBuffer[] sources, int offset, int length)`
Writes the remaining bytes from `length` buffers, starting at `sources[offset]`, into this channel, starting at the channel's current position or its end. Returns the number of bytes written.

`public int write(ByteBuffer source, long start)`
Writes the remaining `bytes` from the `source` buffer into this channel, starting at `start` in the channel. Does not change the channel's current position. Returns the number of bytes written.

`public int read(ByteBuffer destination)`
Reads the remaining bytes from this channel into the remaining positions in `destination` buffer. Returns the number of bytes read or `-1` if buffer not filled.

`public long read(ByteBuffer[] destinations)`
Reads the remaining bytes from this channel into the remaining positions in the buffers in the `destinations` array. Returns the number of bytes read or `-1` if buffers are not filled.

`public long read(ByteBuffer[] destinations, int offset, int length)`
Reads the remaining bytes from this channel into the remaining positions in `length` buffers, starting at `destinations[offset]`. Returns the number of bytes read or `-1` if buffers are not filled.

`public int read(ByteBuffer destination, long start)`
Reads bytes after `start` in this channel into the remaining positions in the `destination` buffer. Returns the number of bytes read or `-1` if buffer is not filled. Does not change channel `position`.

`public MappedByteBuffer map(`
`  FileChannel.MapMode mode, long position, long size)`
Creates a persistent view of `size` bytes of this channel's file, starting at this channel's current `position`, with modes `READ_ONLY`, `READ_WRITE` (mutable view), or `PRIVATE` (copy).

Figure 16.12 Selected FileChannel

methods

The first three read methods start reading at the current channel position, and this position automatically increments as data is copied. The first read method puts data into a single destination buffer, starting at that buffer’s current position. The second read method puts data into a sequence of destination buffers, starting at each one’s current position. The third read method puts data into only those buffers in a specified subarray of an array of buffers. The fourth read method is like the first read method, except it starts reading at the user-specified byte-index, start, and the channel’s internal position does not change. After copying data from the channel to buffer(s), you can copy that data from the buffer(s) using ByteBuffer’s appropriate read method.

The last method in Figure 16.12,

map, is particularly interesting. You invoke this method while the channel is open, and the buffer it creates continues to exist and be accessible after the channel closes. For very large files, using a map improves I/O performance. When you need repeated access to the same file in different contexts, it simplifies your program.

File Channel I/O and Memory Map Example

The ChanneledFileAccess program in Figures 16.13a and 16.13b shows how all this fits together. This program organizes the data in a file like the data in a table—a spreadsheet table or in a relational database table. This program's table has three columns. The first column contains ints. Because an int is 4 bytes, this column is four bytes wide. The second column contains Strings. It is arbitrarily 12 bytes wide. The third column contains doubles.

Because a double is 8 bytes, this column is 8 bytes wide. Thus, each row in the table has a 4-byte int field, followed by a 12-byte String field, followed by an 8-byte double field, for a total of 24 bytes. All data in one row is one *record*. We use a 24-byte ByteBuffer to copy records into and out of the same file channel with the simplest FileChannel methods in Figure 16.12:

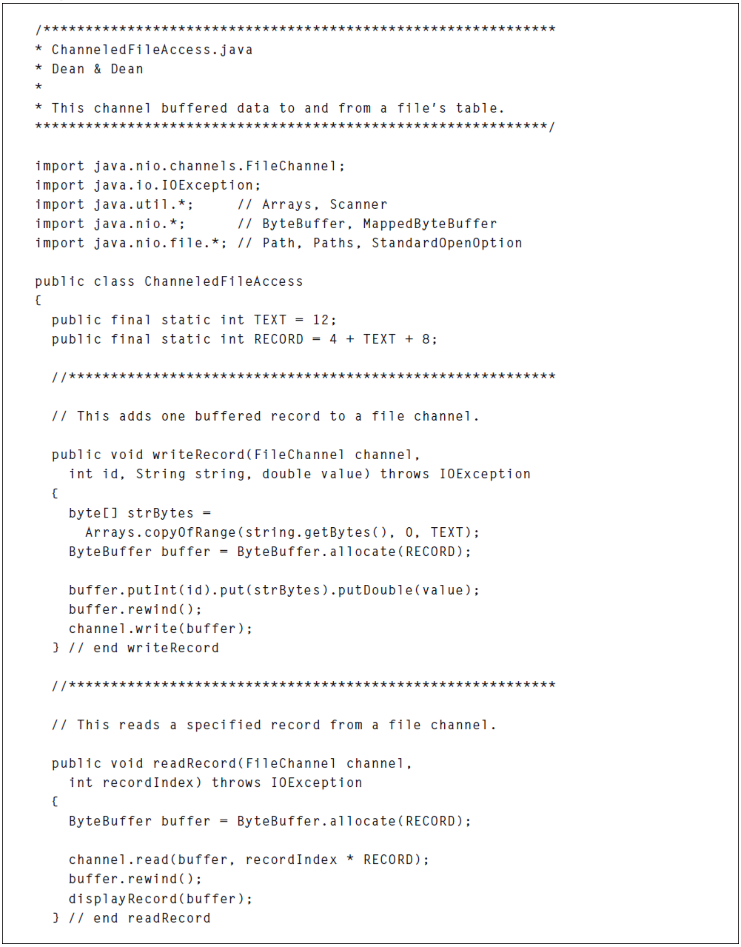
```
write(ByteBuffer source)
read(ByteBuffer destination)
```

This program requires a substantial number of imports. It defines a TEXT constant to establish the text field length and a RECORD constant to establish total record length and the capacity of a buffer that holds exactly one record. Later, the program uses a multiple of the RECORD constant to allocate capacity for another buffer that can hold several records.

The program's writeRecord method writes individual records to a specified channel. This method's parameters are the channel and values for each of the fields in one record. Notice that this method's header includes throws IOException. The first declaration creates an array of bytes whose length equals the length of our record's text field, and it initializes this array with the value of the method's string parameter. The Arrays.copyOfRange method discards bytes beyond 12 if string is too long for the text field. The second declaration creates a buffer whose capacity is RECORD bytes. The first statement after the declarations chains three method calls. The first method call puts the 4-byte ID number into the ID field. This advances the buffer's internal position to 4. The second method call puts the 12-byte string into the text field. This advances the buffer's internal position to 12. The third method call puts the 8-byte double into the value field. The next statement rewinds the buffer. The last statement writes the buffer's bytes into the channel. This channel.write method call is what throws the IOException.

The program's readRecord method reads the record identified by a specified recordIndex. This method's header also includes throws IOException. The declaration creates a buffer whose capacity is RECORD bytes. The first statement after the declaration reads bytes from the channel into the buffer. It starts reading at the channel position given by recordIndex * RECORD, and the total number of bytes it reads equals the buffer's size, which is RECORD. This channel.read method call is what throws the IOException. The next statement rewinds the buffer. The last statement calls the program's displayRecord method to display the record that was read.

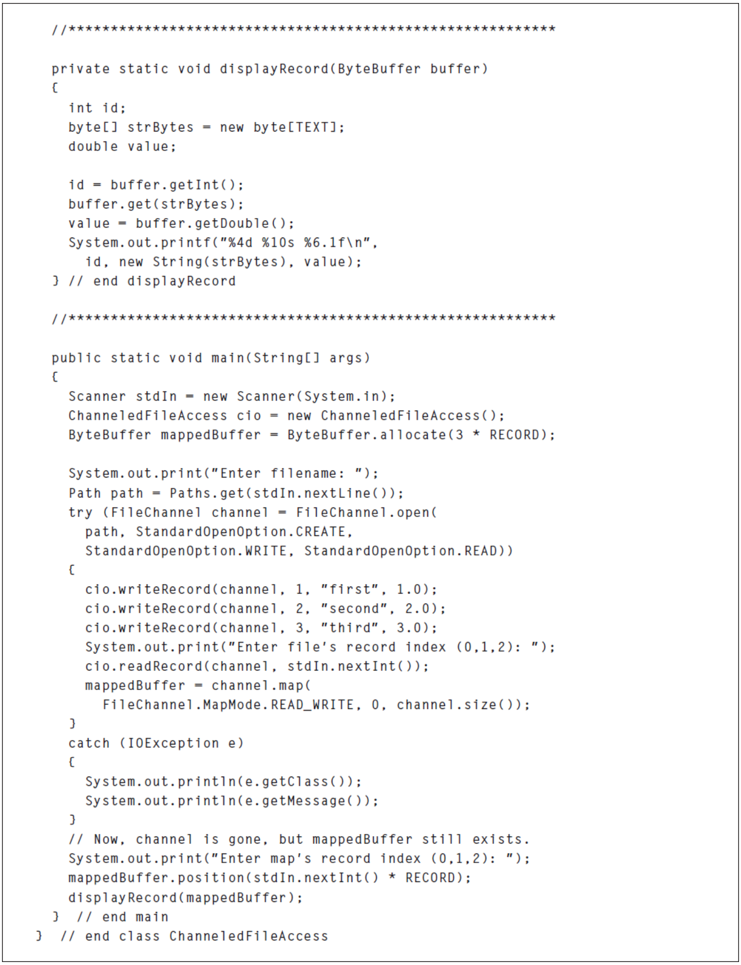
Page 764



```
/****************************************************************
* ChanneledFileAccess.java
* Dean & Dean
*
* This channel buffered data to and from a file's table.
****************************************************************/

import java.nio.channels.FileChannel;
import java.io.IOException;
import java.util.*;     // Arrays, Scanner
import java.nio.*;      // ByteBuffer, MappedByteBuffer
import java.nio.file.*; // Path, Paths, StandardOpenOption

public class ChanneledFileAccess
{
  public final static int TEXT = 12;
  public final static int RECORD = 4 + TEXT + 8;

  //*************************************************************

  // This adds one buffered record to a file channel.

  public void writeRecord(FileChannel channel,
    int id, String string, double value) throws IOException
  {
    byte[] strBytes =
      Arrays.copyOfRange(string.getBytes(), 0, TEXT);
    ByteBuffer buffer = ByteBuffer.allocate(RECORD);

    buffer.putInt(id).put(strBytes).putDouble(value);
    buffer.rewind();
    channel.write(buffer);
  } // end writeRecord

  //*************************************************************

  // This reads a specified record from a file channel.

  public void readRecord(FileChannel channel,
    int recordIndex) throws IOException
  {
    ByteBuffer buffer = ByteBuffer.allocate(RECORD);

    channel.read(buffer, recordIndex * RECORD);
    buffer.rewind();
    displayRecord(buffer);
  } // end readRecord
```

Figure 16.13a ChanneledFileAccess program—part A

Page 765



```
  //**************************************************************

  private static void displayRecord(ByteBuffer buffer)
  {
    int id;
    byte[] strBytes = new byte[TEXT];
    double value;

    id = buffer.getInt();
    buffer.get(strBytes);
    value = buffer.getDouble();
    System.out.printf(”%4d %10s %6.1f\n”,
      id, new String(strBytes), value);
  } // end displayRecord

  //**************************************************************

  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    ChanneledFileAccess cio = new ChanneledFileAccess();
    ByteBuffer mappedBuffer = ByteBuffer.allocate(3 * RECORD);

    System.out.print(”Enter filename: ”);
    Path path = Paths.get(stdIn.nextLine());
    try (FileChannel channel = FileChannel.open(
      path, StandardOpenOption.CREATE,
      StandardOpenOption.WRITE, StandardOpenOption.READ))
    {
      cio.writeRecord(channel, 1, ”first”, 1.0);
      cio.writeRecord(channel, 2, ”second”, 2.0);
      cio.writeRecord(channel, 3, ”third”, 3.0);
      System.out.print(”Enter file’s record index (0,1,2): ”);
      cio.readRecord(channel, stdIn.nextInt());
      mappedBuffer = channel.map(
        FileChannel.MapMode.READ_WRITE, 0, channel.size());
    }
    catch (IOException e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
    // Now, channel is gone, but mappedBuffer still exists.
    System.out.print(”Enter map’s record index (0,1,2): ”);
    mappedBuffer.position(stdIn.nextInt() * RECORD);
    displayRecord(mappedBuffer);
  } // end main
} // end class ChanneledFileAccess
```

Figure 16.13b ChanneledFileAccess program—part B

In Figure 16.13b, we made the program’s displayRecord method a static method so that we can call it directly from main, as well as from readRecord. The displayRecord method prints one record's data by reading exactly RECORD bytes from the buffer parameter, starting at that parameter's current position. This method declares three variables for the three fields in a record. The first statement after the declarations gets the 4-byte integer, id. This advances the buffer's position to the start of the current record's text field. The next statement gets enough of the subsequent buffer's bytes to just fill the 12-byte strBytes array. This advances the buffer's position to the start of the current record's value field. The next statement fills value with an 8-byte double. The last statement prints the specified record's data.

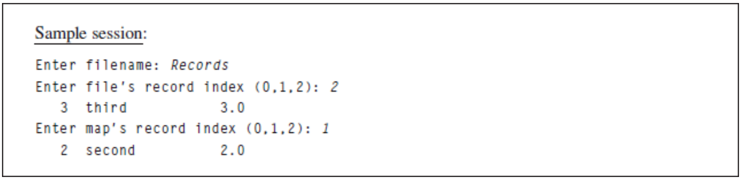
The main method's declarations create a Scanner for keyboard input and a ChanneledFileAccess object to access the program's writeRecord and readRecord methods. They also create a ByteBuffer, which will contain a mapped view of the file's contents that continues to exist after the channel closes.

After the declarations, main asks the user for a filename and uses it to create a Path. Then main asks FileChannel to open a new file channel for both WRITE and READ access. Because this opening method call throws a checked exception, and because we would like later file closing to be automatic, we put this opening method call in a try-with-resources header. Inside the associated try block, main calls writeRecord three times to write three distinct records to the channel. Then it asks the user to specify the index of a particular record in the file and calls the readRecord method to read and display the selected record's data.

While the channel is still open, main calls FileChannel's map method to create a mappedBuffer object that provides a persistent view of the file's data. The map method call throws a checked exception, so it, too, must be inside the try block. Because the preceding calls to our writeRecord and readRecord methods are necessarily also in the try block, the associated catch block also can catch the exceptions these methods throw.

Finally, after the catch block, when the channel is closed, main asks the user to specify the index of a particular record in the mappedBuffer object. Then it calls the displayRecord method to print the data selected from mappedBuffer. The success of this last operation demonstrates the persistence of a mappedBuffer.

Figure 16.13c shows output generated by the ChanneledFileAccess program for particular record index specifications. The first output comes from readRecord's call of the displayRecord method while the channel is still open. In this case, the argument passed to displayRecord is a 24-byte buffer containing only the selected record. The selected record starts at byte 48 in the file, but the 24-byte buffer containing that record enters displayRecord with buffer position equal to zero. The second output comes from main‘s call of the displayRecord method after the channel closes. In this case, the selected record starts at byte 24 in the file, and the 72-byte mapped buffer enters displayRecord with buffer position equal to 24. These record index selections are arbitrary. Each selection could have been any of the three indices, 0, 1, or 2, and if they had been the same, the displayed data would have been the same.



```
Sample session:

Enter filename: Records
Enter file's record index (0,1,2): 2
   3  third          3.0
Enter map's record index (0,1,2): 1
   2  second         2.0
```

Figure 16.13c ChanneledFileAccess Program output

This relatively simple ChanneledFileAccess program demonstrates Java's ability to organize and randomly access file data and to provide persistent views of file data. These capabilities appear even with just the simplest of the FileChannel's read and write methods. If you go back to Figure 16.12 and look at what FileChannel's other read and write methods can do, you'll see that Java provides a great variety of file organization and access alternatives.

16.9 Path, Whole-File, and Directory Operations

Up until now, this chapter has dealt with details—the details of writing data to a file and reading data from a file. Now, we'll back away from details and consider operations on whole files.

In this section, we'll look at alternate ways to describe the path to a file. We'll look at ways to copy or move a file from one directory to another. And we'll look at ways to display directory contents. Later sections will show how to search the directory tree for particular file names and how to display directory contents with GUI.

Defining and Manipulating Paths

To perform any of these operations, we must tell the computer where each file of interest is. We do that by creating a path that leads to the file. Then we can use the path to specify the file. As you have seen, the simplest way to create a Path is to use the Files.get method in the java.nio.file package in the Java API:

Path path = Files.get("*path-to-directory-or-file*");

The *path-to-directory-or-file* string can have any of several different forms. It can be an *absolute path*. An absolute path starts at the root of the directory hierarchy (C:/ in Windows or just / in Unix). It progresses down the directory hierarchy with each subdirectory followed by a forward slash. If it's a path to a directory, that directory's name is the name just before the last forward slash. If it's a path to a file, that file's name is the name just after the last forward slash. It can be a *relative path*, which starts at the current directory.3 The relative path to the current directory is a single dot (.). The relative path to another file named "sisterSally" in the current directory is just the name of that other file (sisterSally). Assuming the current directory contains a subdirectory called, "sisterSonia," the relative path to a file named "nieceNedra" in that subdirectory is

sisterSonia/nieceNedra

The relative path to the directory above the current directory—the parent directory—is a pair of dots (..). The relative path the directory above that—the grandparent directory—is two pairs of dots (../..). And so forth. Assuming the parent directory contains another directory called “auntAgnes,” the relative path to a file named “cousinCora” in that other directory is

```
../auntAgnes/cousinCora
```

Suppose you have a path created by using just the name of a file in the current directory. That’s a relative path. Let’s call it pathR. You can obtain the corresponding absolute path, pathA, like this:

```
Path pathA = pathR.toAbsolutePath();
```

Given two absolute paths, pathA1 and pathA2, you can obtain the relative path from the end of pathA1 to the end of pathA2 like this:

```
Path path1_to_path2 = pathA1.relativize(pathA2);
```

Given an absolute (or a relative) Path called path, to find the total number of elements after the root (or current directory), use

```
path.getNameCount();
```

To identify the subpath from the element after int start through int end, use

```
path.subpath(start, end);
```

To obtain the combination of path1 followed by path2, use

```
Path pathComb = path1.resolve(path2);
```

Creating, Moving, Copying, and Deleting Files

The file designated by a path may not exist when a path is created. The

Files.exists(path), method call returns true if a file already exists at the location specified by the path, and if you have permission to access that file. To create a new directory, use Path.createDirectory(path). To create a new regular file, you can use Files.createFile(path) as described in Section 15.8, but usually you'll just open a new file for writing, as illustrated in Chapter 15 and several times in this chapter. Use Files.isDirectory(path) to see if path leads to a directory. Use Files.isRegularFile to see if path leads to a regular file.

To move or copy an existing directory or file from path1 to path2, in a try block, use

Files.move(path1, path2, *option(s)*);

or

Files.copy(path1, path2, *option(s)*);

There may be zero or more options. If

there are no options, and path2 refers to an existing file, either a move or a copy method call throws a FileAlreadyExistsException. To replace an existing file of the same name, include the option

StandardCopyOption.REPLACE_EXISTING

Java API documentation describes other possible options.

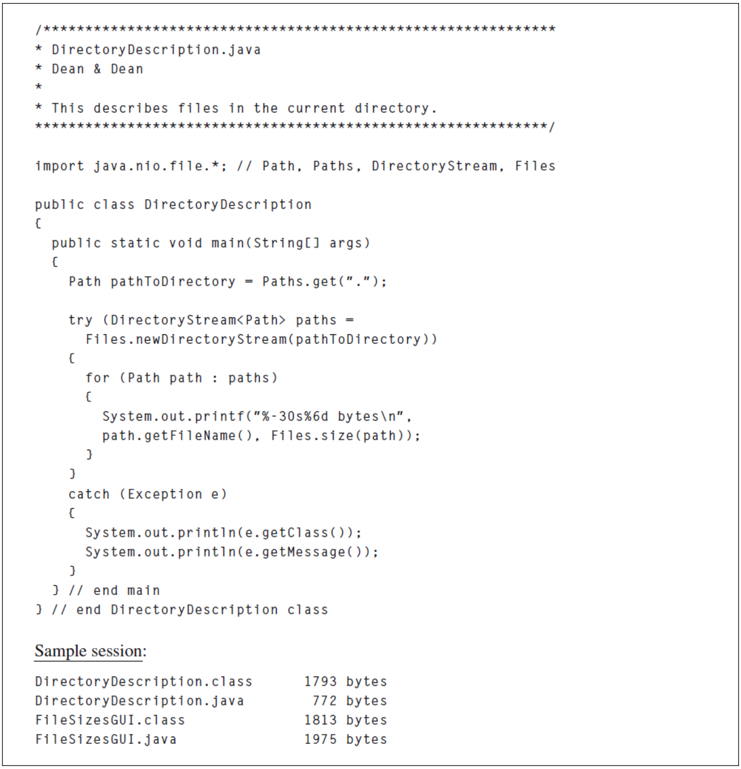
To delete a file referred to by path, in a try block, use

Files.delete(path);

If path refers to a nonexistent file, this call throws a NoSuchFileException. If path refers to a directory that is not empty, this call throws a DirectoryNotEmptyException. Before you can delete a directory, you must first delete or remove all the files it contains.

Describing a Directory’s Contents

Figure 16.14 shows how you might display the contents of a directory. The first statement in the main method creates a relative path to the current directory. The statement in the try-with-resources header uses this path to create a DirectoryStream, which contains information about what's in the directory at the end of the specified path. The printf statement in the try block displays filename and file size (length in bytes), as shown in the sample session below the code. Size is just one of several file attributes. Java API documentation describes other file attributes and explains how to access and modify them.



```
/*************************************************************
* DirectoryDescription.java
* Dean & Dean
*
* This describes files in the current directory.
*************************************************************/

import java.nio.file.*; // Path, Paths, DirectoryStream, Files

public class DirectoryDescription
{
  public static void main(String[] args)
  {
    Path pathToDirectory = Paths.get(”.”);

    try (DirectoryStream<Path> paths =
      Files.newDirectoryStream(pathToDirectory))
    {
      for (Path path : paths)
      {
        System.out.printf(”%-30s%6d bytes\n”,
        path.getFileName(), Files.size(path));
      }
    }
    catch (Exception e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end DirectoryDescription class
```

Sample session:

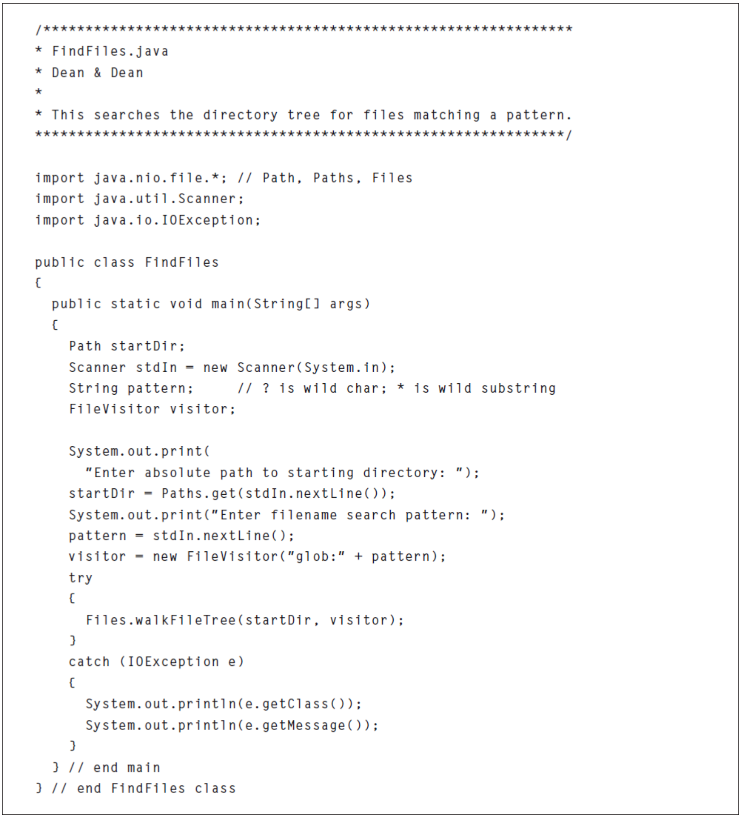
```
DirectoryDescription.class      1793 bytes
DirectoryDescription.java        772 bytes
FileSizesGUI.class              1813 bytes
FileSizesGUI.java               1975 bytes
```

Figure 16.14 DirectoryDescription program

16.10 Walking a Directory Tree

There’s another way to explore a file’s environment. That’s by using the interfaces, classes, and enums in the java.nio.file package. To illustrate this other approach, this section presents a program that searches a user-specified subtree of your computer's file directory for files whose names conform to a user-specified text pattern. The program's output shows exactly where the files it finds are located.

Figure 16.15a shows the driver of this FindFiles program. The first declared variable is for the root of the user-specified subtree. It must be an absolute path, starting with a forward slash and ending with the name of the root of the desired subtree, like this: */…/sub-tree root*. The second declared variable is for keyboard input. The third declared variable is for the user-specified pattern that determines which files will be identified. This text string should match all desired filenames, with a ? substituted for each character that could be anything and/or a * substituted for each substring that could be anything. For example, the entry, *.java, tells the program to find all Java source code files. The last declared variable, visitor, is a special object that does most of the program's work.



```
/*******************************************************************
* FindFiles.java
* Dean & Dean
*
* This searches the directory tree for files matching a pattern.
*******************************************************************/

import java.nio.file.*; // Path, Paths, Files
import java.util.Scanner;
import java.io.IOException;

public class FindFiles
{
  public static void main(String[] args)
  {
    Path startDir;
    Scanner stdIn = new Scanner(System.in);
    String pattern;     // ? is wild char; * is wild substring
    FileVisitor visitor;

    System.out.print(
      "Enter absolute path to starting directory: ");
    startDir = Paths.get(stdIn.nextLine());
    System.out.print("Enter filename search pattern: ");
    pattern = stdIn.nextLine();
    visitor = new FileVisitor("glob:" + pattern);
    try
    {
      Files.walkFileTree(startDir, visitor);
    }
    catch (IOException e)
    {
      System.out.println(e.getClass());
      System.out.println(e.getMessage());
    }
  } // end main
} // end FindFiles class
```

Figure 16.15a FindFiles program driver

After the program prompts for and reads the two user specifications, it creates the visitor object. The "glob:" part of the argument passed to the FileVisitor constructor tells the computer to treat ? and * as wildcards whenever it finds them in the pattern part of the constructor's argument.4 Then the program calls the walkFileTree method, which throws a checked exception. Figure 16.15b shows output produced by a typical execution. Whenever the program finds a matching file, it displays that file's name and the number of bytes in that file. Names without reported bytes are directory names.

Now, let's look at heart of the program—the FileVisitor class in Figures 16.16a and 16.16b. Figure 16.16a contains the imports, declarations, and constructor. You can see that the Java API code that does most of the work comes from various places in the java.nio.file package. And you can see that our FileVisitor class is actually just an extension of Java's SimpleFileVisitor class. Our class's first instance variable is a PathMatcher, which compares the user's specified pattern with the name of each file it encounters. Our class's second instance variable, tab, controls the output's line indentation. The constructor generates a PathMatcher with the user-specified pattern.

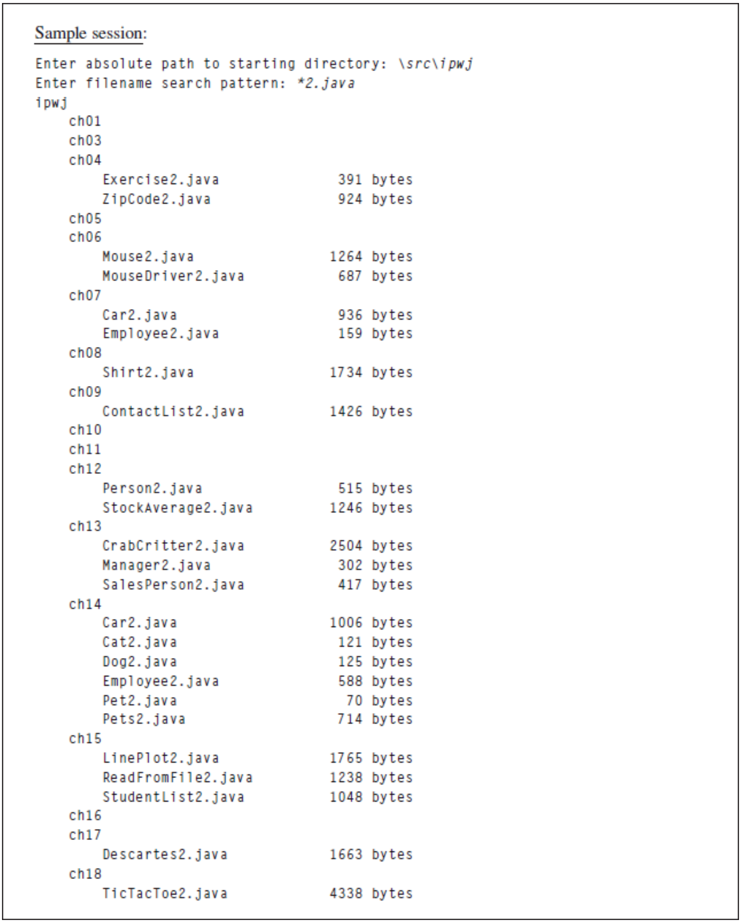
Figure 16.15b shows that the program visits files in the specified subtree in what mathematicians call a *depth-first traverse*. That's a formal term for what Java designers call "walking the tree." The program actually visits all files in the subtree, but it generates printouts only for directories and for regular files whose names match the user's specified pattern. What you see in Figure 16.15b indicates that the program does something in three situations: (1) when it encounters a new directory going down the hierarchy, (2) when it encounters a regular file, and (3) when it returns to a directory above after visiting all its contained files and subdirectories.

Figure 16.16b contains three methods that define what this program does in each of these three situations. Its methods override three corresponding methods in the extended SimpleFileVisitor class to produce the particular results that we want. The @Override notation just above each method's heading helps the compiler confirm that the method does indeed correspond to a method in the inherited class. As it walks the tree, when the program encounters any new directory going down the hierarchy, it calls the preVisitDirectory method. After producing appropriate indentation, that method prints that directory's name, and then it increments tab to increase subsequent indentation. When the program encounters a file whose filename matches the user's specification, it calls the visitFile method. After producing appropriate indentation, that method prints that file's name and size in bytes. When the program returns to a directory after visiting all its files and subdirectories, it calls the postVisitDirectory method. All it does is decrement tab to decrease subsequent indentation.

Suppose all you want to see is the directory structure, not any of the files in that structure. For this result, just run the program with nothing but a simple carriage return (Enter) for the pattern specification.

Suppose that what you want to see is a listing of matching files only, with each file identified as the total path from the directory's root to that file. For this result, do not override the preVisitDirectory and postVisitDirectory methods. That is, delete or comment out these two methods in Figure 16.16b. Delete the tab variable and replace the for loop in the visitFile method with this statement:

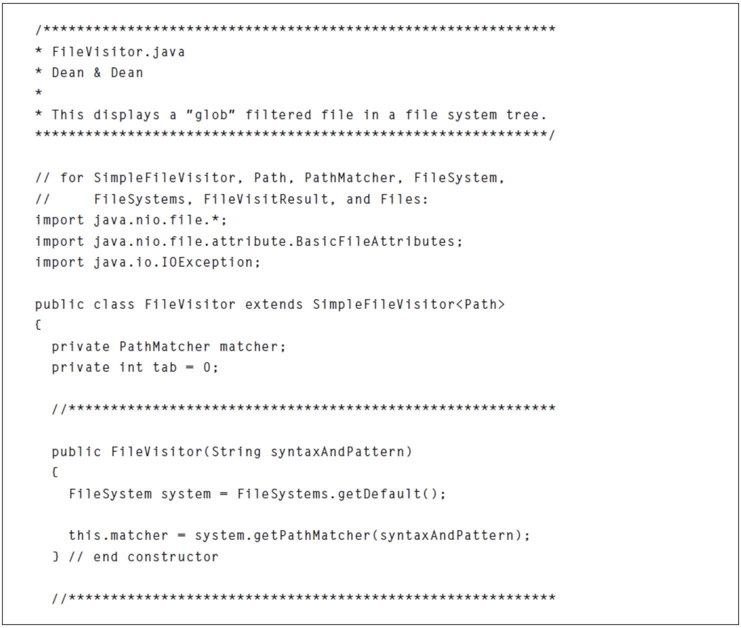
```
name = name.toAbsolutePath();
```



Sample session:

```
Enter absolute path to starting directory: \src\ipwj
Enter filename search pattern: *2.java
ipwj
    ch01
    ch03
    ch04
        Exercise2.java              391 bytes
        ZipCode2.java               924 bytes
    ch05
    ch06
        Mouse2.java                1264 bytes
        MouseDriver2.java           687 bytes
    ch07
        Car2.java                   936 bytes
        Employee2.java              159 bytes
    ch08
        Shirt2.java                1734 bytes
    ch09
        ContactList2.java          1426 bytes
    ch10
    ch11
    ch12
        Person2.java                515 bytes
        StockAverage2.java         1246 bytes
    ch13
        CrabCritter2.java          2504 bytes
        Manager2.java               302 bytes
        SalesPerson2.java           417 bytes
    ch14
        Car2.java                  1006 bytes
        Cat2.java                   121 bytes
        Dog2.java                   125 bytes
        Employee2.java              588 bytes
        Pet2.java                    70 bytes
        Pets2.java                  714 bytes
    ch15
        LinePlot2.java             1765 bytes
        ReadFromFile2.java         1238 bytes
        StudentList2.java          1048 bytes
    ch16
    ch17
        Descartes2.java            1663 bytes
    ch18
        TicTacToe2.java            4338 bytes
```

Figure 16.15b Typical FindFiles program output



```
/*************************************************************
* FileVisitor.java
* Dean & Dean
*
* This displays a "glob" filtered file in a file system tree.
*************************************************************/

// for SimpleFileVisitor, Path, PathMatcher, FileSystem,
//     FileSystems, FileVisitResult, and Files:
import java.nio.file.*;
import java.nio.file.attribute.BasicFileAttributes;
import java.io.IOException;

public class FileVisitor extends SimpleFileVisitor<Path>
{
  private PathMatcher matcher;
  private int tab = 0;

  //**********************************************************

  public FileVisitor(String syntaxAndPattern)
  {
    FileSystem system = FileSystems.getDefault();

    this.matcher = system.getPathMatcher(syntaxAndPattern);
  } // end constructor

  //**********************************************************
```

Figure 16.16a FileVisitor class—part A

Suppose you want to truncate or terminate the search. In Figure 16.16b, with appropriate code adjustments, use the following code:

return FileVisitResult.SKIP_SIBLINGS

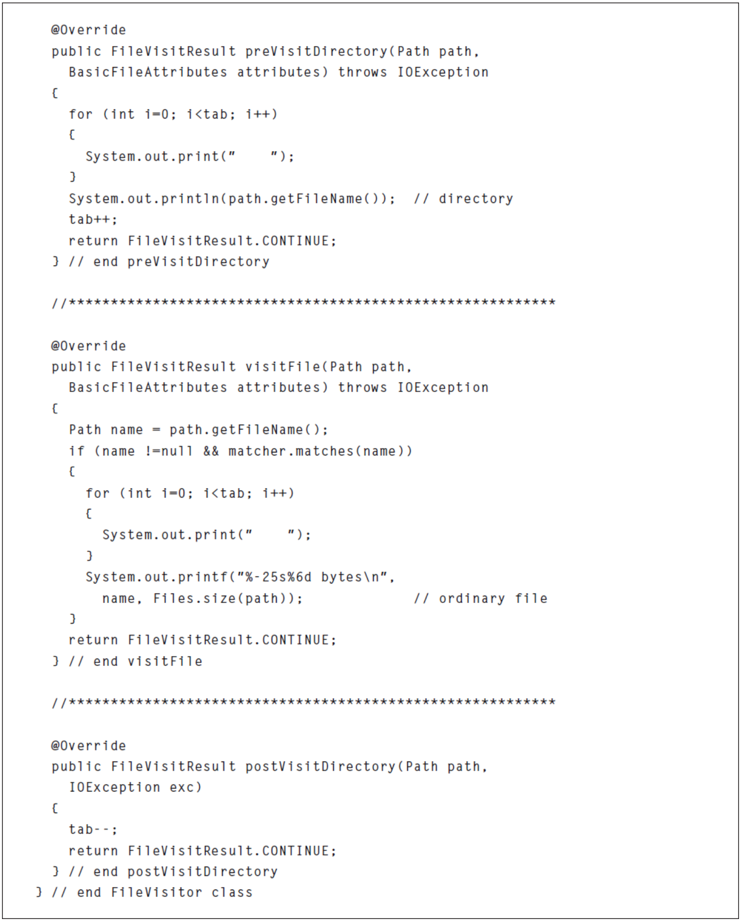
or

return FileVisitResult.SKIP_SUBTREE

or

return FileVisitResult.TERMINATE

One of the end-of-chapter exercises asks you to use the SKIP_SUBTREE return option in a modification of our FindFiles program that displays files and file sizes in one directory only.



```
    @Override
    public FileVisitResult preVisitDirectory(Path path,
      BasicFileAttributes attributes) throws IOException
    {
      for (int i=0; i<tab; i++)
      {
        System.out.print("    ");
      }
      System.out.println(path.getFileName());  // directory
      tab++;
      return FileVisitResult.CONTINUE;
    } // end preVisitDirectory

    //***********************************************************

    @Override
    public FileVisitResult visitFile(Path path,
      BasicFileAttributes attributes) throws IOException
    {
      Path name = path.getFileName();
      if (name !=null && matcher.matches(name))
      {
        for (int i=0; i<tab; i++)
        {
          System.out.print("    ");
        }
        System.out.printf("%-25s%6d bytes\n",
          name, Files.size(path));             // ordinary file
      }
      return FileVisitResult.CONTINUE;
    } // end visitFile

    //***********************************************************

    @Override
    public FileVisitResult postVisitDirectory(Path path,
      IOException exc)
    {
      tab--;
      return FileVisitResult.CONTINUE;
    } // end postVisitDirectory
  } // end FileVisitor class
```

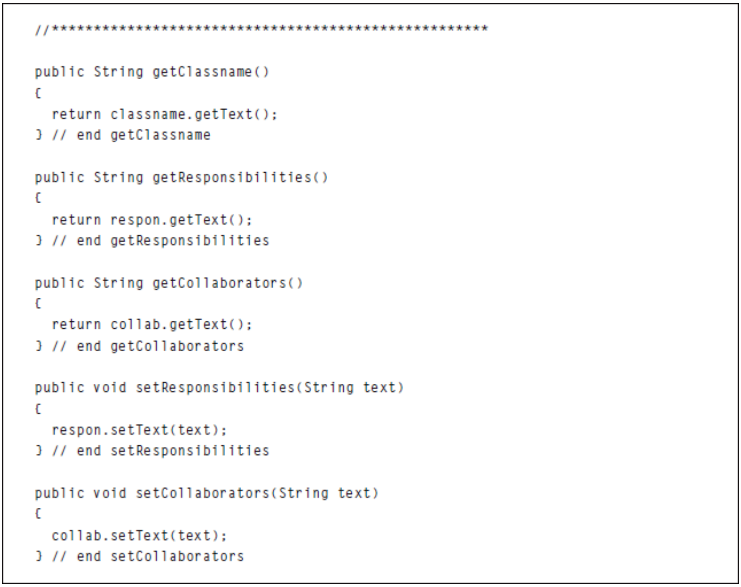
Figure 16.16b FileVisitor class—part B

16.11 GUI Track: Final Iteration of Problem Solving with CRC Cards (Optional)

This section describes the final iteration of the CRC_Card program introduced in Chapter 8's GUI section and improved in Chapter 10's GUI section. We enhance Chapter 8's CRC_Card class by adding get and set methods to make a new CRC_Card2 class. We add an easily serializable CardData class to facilitate file operations. And we convert Chapter 10's CRCDriver2 class to CRCDriver3 with enhancements that enable the user to save any card's user-entered information to a file and recover it at any later time.

Figure 16.17 contains the code for this iteration's enhancements of the CRC_Card class described in Section

8.16 and presented in Figure 8.19. These enhancements add get and set methods to enable other code to read from all three of CRC_Card2's state variables and write to its respon and collab state variables. (The constructor writes to the classname variable.) This provides program access to all user-entered information.

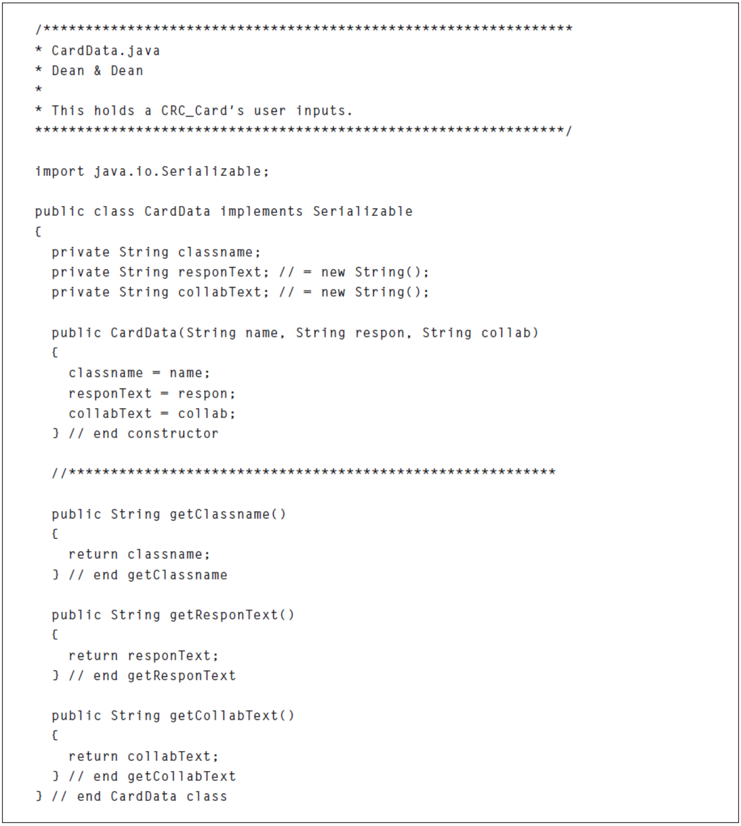


```
//****************************************************

public String getClassname()
{
  return classname.getText();
} // end getClassname

public String getResponsibilities()
{
  return respon.getText();
} // end getResponsibilities

public String getCollaborators()
{
  return collab.getText();
} // end getCollaborators

public void setResponsibilities(String text)
{
  respon.setText(text);
} // end setResponsibilities

public void setCollaborators(String text)
{
  collab.setText(text);
} // end setCollaborators
```

Figure 16.17 Additions to CRC_Card class in Figure 8.19 for this section's CRC_Card2 class

The CardData class in Figure 16.18 uses the get and set methods defined in

Figure 16.17’s CRC_Card2 class in order to create a serializable object that we can save in a file. For a class to be serializable, it must implement the Serializable interface, and all of its content must also implement the Serializable interface. That’s easy for CardData because its content consists of String objects, and the String class already implements Serializable. If we tried to save a complete CRC_Card2, we’d be frustrated. Serializing an empty Stage is straightforward, but serializing all subordinate containers and components would be tedious. It’s easier to copy a CRC_Card2’s distinctive user-entered text into a CardData object and put that CardData object in a file. Then later we can retrieve that file, instantiate a new CRC_Card2 with the saved classname in its constructor, and use the set methods in Figure 16.17 to restore the other user-entered text.



```
/***************************************************************
* CardData.java
* Dean & Dean
*
* This holds a CRC_Card's user inputs.
***************************************************************/

import java.io.Serializable;

public class CardData implements Serializable
{
  private String classname;
  private String responText; // = new String();
  private String collabText; // = new String();

  public CardData(String name, String respon, String collab)
  {
    classname = name;
    responText = respon;
    collabText = collab;
  } // end constructor

  //**********************************************************

  public String getClassname()
  {
    return classname;
  } // end getClassname

  public String getResponText()
  {
    return responText;
  } // end getResponText

  public String getCollabText()
  {
    return collabText;
  } // end getCollabText
} // end CardData class
```

Figure 16.18 Easily serializable CardData class for filing user-entered CRC_Card information

As part of the CRC_Card program's final modifications, we convert the CRCDriver2 class described in Section 10.12 into a CRCDriver3 class. We describe the simpler modifications next, and present the more extensive ones later.

For CRCDriver3’s imports, add a comment indicating that the previous import javafx.stage.*; also provides access to the prewritten FileChooser class. Then add the following additional import:

```
// ObjectOutputStream, FileOutputStream,
// File, ObjectInputStream, FileInputStream, EOFException
import java.io.*;
```

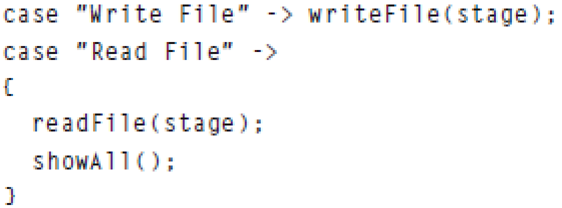
In the start method, change two instances of CRC_Card to CRC_Card2 and change selectTask() to selectTask(stage). Then substitute private void selectTask(Stage stage) for the header of the selectTask method. The stage reference enables the program to block other operations while file operations are in progress.

In the selectTask method, add “Write

File” and “Read File” choices to the task variable’s ChoiceDialog object:

```
ChoiceDialog<String> task = new ChoiceDialog<>("Quit",
"Quit", "Add", "Show All", "Print Card",
"Write File", "Read File");
```

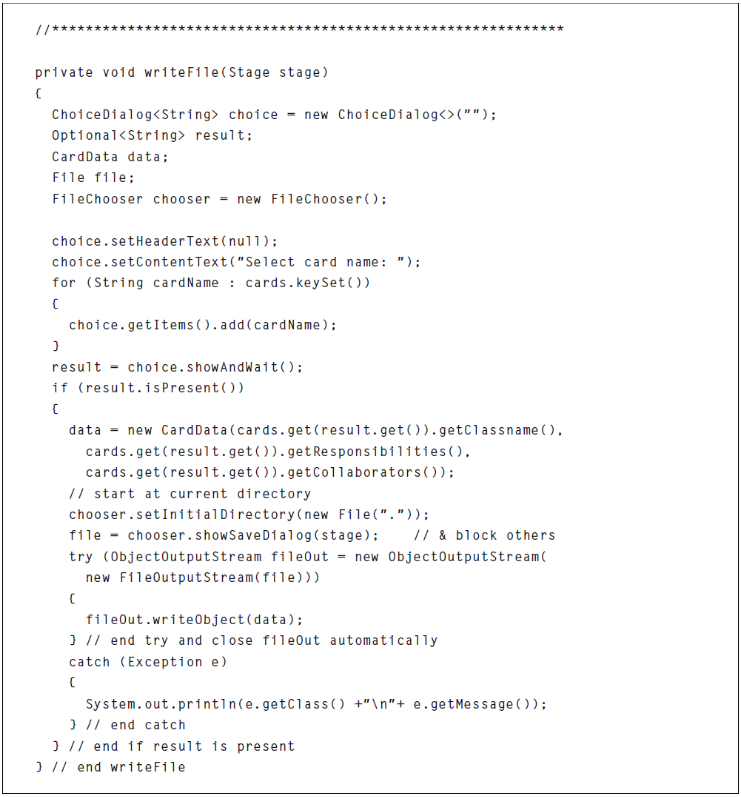
And in the switch statement between the “Print Card” and “Quit” case clauses, insert these case clauses:



```
case "Write File" -> writeFile(stage);
case "Read File" ->
{
  readFile(stage);
  showAll();
}
```

To implement the additional write and read choices, we add the helper methods writeFile and readFile to the enhanced CRCDriver3 class. Figure 16.19 presents the code for the new writeFile method. In the method’s first two declarations, we create a ChoiceDialog to help the user choose the card to save and an Optional variable to return that choice. We then instantiate a FileChooser object that provides access to (probably familiar) platform-dependent windows that help users access and manipulate their computer's files.

The statements immediately after the declarations configure the local ChoiceDialog to list all cards in CRCDriver3's LinkedHashMap cards instance variable. After the user makes a selection and a result is present, we construct and populate a CardData object with all of that card's user-entered information. Then we call the FileChooser object's showSaveDialog method, which displays a window that enables the user to select a directory and filename and perform a save operation. Think of this window as a highly specialized and sophisticated version of the Dialog windows you have learned to love. The user can select an existing file to update, enter the previously selected card name, or enter a filename like the selected card name but with an appended or incremented version number.

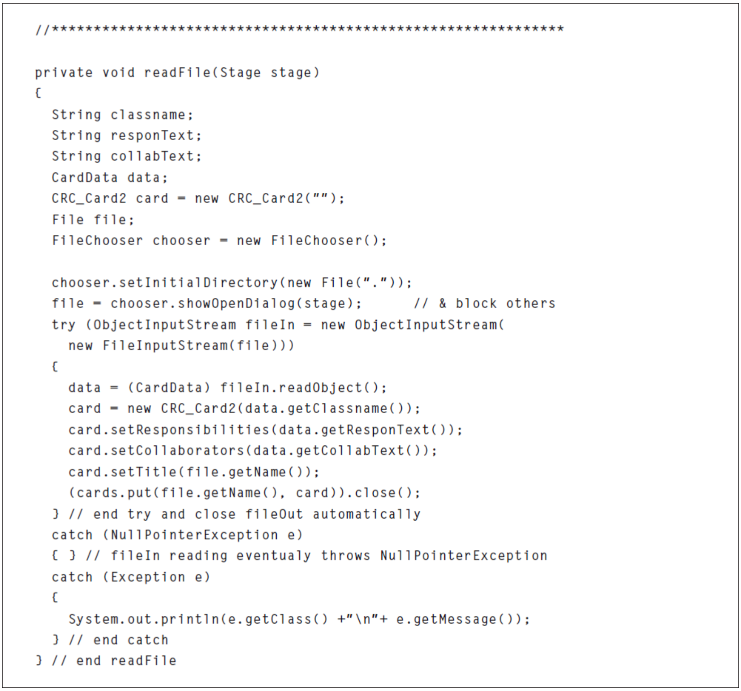


```
//*************************************************************

private void writeFile(Stage stage)
{
  ChoiceDialog<String> choice = new ChoiceDialog<>("");
  Optional<String> result;
  CardData data;
  File file;
  FileChooser chooser = new FileChooser();

  choice.setHeaderText(null);
  choice.setContentText("Select card name: ");
  for (String cardName : cards.keySet())
  {
    choice.getItems().add(cardName);
  }
  result = choice.showAndWait();
  if (result.isPresent())
  {
    data = new CardData(cards.get(result.get()).getClassname(),
      cards.get(result.get()).getResponsibilities(),
      cards.get(result.get()).getCollaborators());
    // start at current directory
    chooser.setInitialDirectory(new File("."));
    file = chooser.showSaveDialog(stage);    // & block others
    try (ObjectOutputStream fileOut = new ObjectOutputStream(
      new FileOutputStream(file)))
    {
      fileOut.writeObject(data);
    } // end try and close fileOut automatically
    catch (Exception e)
    {
      System.out.println(e.getClass() +"\n"+ e.getMessage());
    } // end catch
  } // end if result is present
} // end writeFile
```

Figure 16.19 New writeFile method added to CRCDriver2 class for CRCDriver3 class

When the user clicks the file save window’s Save button, showSaveDialog instantiates a File object, which stores a filename and the path where the file is stored within the current file system, and returns a reference to the File object. The writeFile method then assigns that returned reference to a file variable. Then, in a try-with-resources header, we create a FileOutputStream to write data into memory at file's location. Still in the header, we create an ObjectOutputStream fileOut that inserts a serialized object into the just-created FileOutputStream. Finally, in the try block, the "ready-to-go" fileOut object calls its writeOut method with our serialized data object as its input argument.

Figure 16.20 presents the code for our new readFile method. As in our previous writeFile method, the last declaration creates a FileChooser chooser. But this time chooser calls its showOpenDialog method to display the current directory in the local computer’s standard openDialog window. Typically, this window allows the user to navigate to any other directory in the computer, and it allows the user to select any displayed filename. Assuming the user selects a previously saved CardData file, when the user clicks the openDialog window’s Open button, showOpenDialog instantiates a File object. The writeFile method then assigns that File object’s reference to a file variable. Then, in a try-with-resources header, we create a FileInputStream to read data from memory at file’s location. Still in the header, we create an ObjectInputStream fileIn that extracts a serialized object from the just-created FileInputStream.



```
//*************************************************************

private void readFile(Stage stage)
{
  String classname;
  String responText;
  String collabText;
  CardData data;
  CRC_Card2 card = new CRC_Card2("");
  File file;
  FileChooser chooser = new FileChooser();

  chooser.setInitialDirectory(new File("."));
  file = chooser.showOpenDialog(stage);      // & block others
  try (ObjectInputStream fileIn = new ObjectInputStream(
    new FileInputStream(file)))
  {
    data = (CardData) fileIn.readObject();
    card = new CRC_Card2(data.getClassname());
    card.setResponsibilities(data.getResponText());
    card.setCollaborators(data.getCollabText());
    card.setTitle(file.getName());
    (cards.put(file.getName(), card)).close();
  } // end try and close fileOut automatically
  catch (NullPointerException e)
  { } // fileIn reading eventualy throws NullPointerException
  catch (Exception e)
  {
    System.out.println(e.getClass() +"\n"+ e.getMessage());
  } // end catch
} // end readFile
```

Figure 16.20 New readFile method added to CRCDriver2 class for CRCDriver3 class

Now we're ready to go. Like file writing, file reading must occur inside a try block. The first try-block statement gives our data variable a reference to a CardData object that contains the desired file information. The next try-block statement creates a new CRC_Card2 card with data's classname, the next two try-block statements transfer the other data information to card's other state variables. The statement after that makes card's title equal to the current name of the selected file. The final try-block statement puts the newly created card into the LinkedHashMap. LinkedHashMap's put method replaces any identically named component and returns a reference to the replaced component (if any). In a chaining operation, this statement then uses that returned reference to delete any display of the replaced object to avoid its being left on the screen when the program terminates.

The try block's first fileIn.readObject operation throws a NullPointerException when it reaches the end of the stream's data, and we provide an extra catch block to discard it. Happily, this particular exception throwing does not prevent subsequent try-block statements from executing.

Summary

- Most of the file-transfer classes you'll need are in the java.io, java.nio, java.nio.files, and java.nio.channels packages.
- To read text from a simple local file, supply a String filename as the argument in a Paths.get method call to obtain a Path description of the file's location. Then use this Path as the argument in a new Scanner, and use familiar Scanner methods to read the file's data.
- To write text to a simple local file, instantiate a PrintWriter object with a String filename as its argument. Then use PrintWriter methods like println to write data to the file.

- You can use simple text file I/O to translate plain-text information into HTML format for a web page.

- You can use a Scanner with an InputStream from a URLConnection to read the HTML text version of what's on a remote website.

- You can store whole objects in a file as a stream of bytes, provided those objects and all their component objects implement the Serializable interface.

- To write an object to a file, construct a FileOutputStream with a String filename as the argument. Then with that stream as the argument, construct in a new ObjectOutputStream. Then use ObjectOutputStream's writeObject method to write the object's data to the file.

- To read an object from a file,

construct a FileInputStream with a String filename as the argument. Then with that stream as the argument, construct a new ObjectInputStream. Then use ObjectInputStream’s readObject method to read the object’s data from the file.

- To specify a character set, import java.nio.charset.Charset, and use:

  character-set = Charset.defaultCharset(), or something like this:

  character-set = Charset.forName("US-ASCII")

- With large text files, write through a buffer by opening the file with

  new PrintWriter(Files.newBufferedWriter

  Paths.get(*filename*), *character-set*, *open-option(s)*))

Then write the text with methods like println.

- With large text files, read through a buffer by opening the file with

  Files.newBufferedReader(

  Paths.get(*filename*), *character-set*))

  Then read the text with a method like readLine.

- ByteBuffer's put and get methods copy any single primitive or an array of bytes to or from the next available buffer position or an index number of primitive elements from the start of the buffer.

- To copy an array of primitive values to or from a ByteBuffer's underlying array, assign the returned value from ByteBuffer's as *Wrapper* Buffer to a corresponding *Wrapper* Buffer. Then use *Wrapper* Buffer's put and get array methods to access the data in ByteBuffer's underlying array.

- FileChannel methods organize heterogeneous data into sequences of ByteBuffers or arrays of ByteBuffers. To open FileChannel channel for either input or output, use:

  channel = FileChannel.open(Path path, OpenOption... options)

  where ... means there can be any number of options, including none. Java API options are enums like java.nio.file.StandardOpenOption.READ

- The return from FileChannel's method,

map(FileChannel.MapMode mode, long position, long size)

is a nonvolatile connection to file data, which persists beyond FileChannel closure.

- The Java API supplies many useful ways to manipulate paths and describe the files in a directory.

- With an extension of Java's SimpleFileVisitor, you can search a directory tree for files that match a "glob" pattern, which includes ? and * wildcards. To do this, use the static method call, Files.walkFileTree(startDirectory, fileVisitor).

Review Questions

§16.2 Simple Text-File Example—

HTML File Generator

1. Assuming the object that manages output will be called writer, write a statement that opens a text file called dogs.html for output by println statements.

2. Where do the `<h1>` and `</h1>` tags go in an HTML file?

§16.4 Object File I/O

3. To be file-writable and file-readable, an object must be an instance of a class that implements ____________.

4. Write a statement that opens a file for input of objects and assigns a reference to the connection to objectIn. Assume the file is in the current directory, with the name automobiles.data.

5. Assuming you have made the declaration, TestObject testObject; you can assign the value returned by ObjectInputStream’s readObject method directly to testObject. (T / F)

§16.5 Character Sets and File-Access Options

6. Given the filename “TradingPartners,” write a try-with-resources header that opens a file for writing text with a PrintWriter called fileOut and stores that text using the UTF-16 character set.

7. Given the filename “Suppliers,” write a try-with-resources header that (1) opens a text file for reading with a Scanner named fileIn, and (2) interprets the file’s data using the UTF-16 character set.

8. What Java API package contains Enum StandardOpenOption?

§16.6 Buffered Text File I/O

9. In a Java interface, what does the ... notation (an ellipsis) in the parameter specification

OpenOption... options

say about the arguments you should supply to the constructor or method?

10. When a text file is large, for good performance, what type of file handler should you use?

§16.7 Primitive Buffers with Random Access

11. What does a flip() method call do?

12. What is the difference between the ByteBuffer methods, get() and get(int index)?

13. ByteBuffer's putDouble(double value) method inserts the 8-byte value into the buffer starting at the current position and increments position to just after that inserted value. (T / F)

14. Given int[] integers and ByteBuffer buffer, write a statement that copies integers[3] into buffer, starting at buffer's current position.

§16.8 Channel I/O and Memory-Mapped Files

15. Assuming import java.nio.file.*; and an existing file called "CanadianFriends," write a Java statement that declares and opens a read-only FileChannel to that file.

16. Given a FileChannel called channel and a ByteBuffer called buffer whose capacity and limit equal channel.size(), write a statement that maps the entire file into the buffer for read-only access.

§16.9 The File Class

17. Write code for a try-with-resources header that creates a stream containing information about what's in the parent of the current directory.

§16.10 Walking a Directory Tree

18. What is a "glob"?

Exercises

---

1. [after §16.2] HTML coding-style conventions require that all `<p>` start tags have an accompanying `</p>` end tag. Edit the book's HTMLGenerator.java program so that `</p>` end tags are inserted at the bottom of each paragraph. In addition, edit the program so that each text line inside a paragraph is indented by two space characters (" ").

Do not allow a `</p>` end tag to be generated when there's no accompanying `<p>` start tag (start and end tags must always be partnered). Your program should be robust. In particular, it should handle the case where there's only a title and no paragraphs at all.

2. [after §16.4] Piggyback off of the code in the previous chapter's StudentList class so that objects from that class can be written to a file. Provide an updated StudentList class heading so that the class supports serialization. Using this chapter's WriteObject class as a guide, implement a StudentListFileWriter class with a main method that does the following.

- Instantiate an output file object where the filename comes from user input.

- Instantiate a StudentList object with student names Caiden, Jordan, Kelsey, Max, Jack, and Shelly.
- Write the StudentList object to the file.
- Call the remove method to remove Kelsey from the StudentList object.
- Call the file's reset method.
- Append the updated StudentList object to the file.

If you want to test your solution to this exercise, do the next exercise.

3. [after §16.4] Using this chapter's ReadObject class as a guide, implement a StudentListFileReader class that reads and displays all objects in the file created in the previous exercise.

4. [after §16.5] Use Oracle's Java

API documentation to help you describe each of the following StandardOpenOption constants: APPEND, CREATE, READ, TRUNCATE_EXISTING, and WRITE.

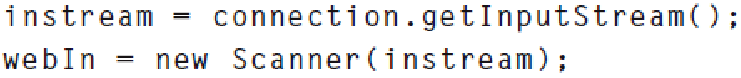
5. [after §16.6] Java's PrintWriter class is more versatile than the BufferedWriter class because it has more methods. Modify this chapter's BufferedWriteToFile program so that instead of using BufferedWriter's write method, it uses PrintWriter's println method. Create a PrintWriter object by using the current program's BufferedWriter object as the argument in a PrintWriter constructor.

6. [after §16.6] Java's Scanner class is more versatile than the BufferedReader class because it has more methods. Modify this chapter's BufferedReadFromFile program so that instead of using BufferedReader's ready and readLine methods, it uses Scanner's hasNextLine and nextLine methods. In the try-with-resources header create a Scanner object by using the current program's BufferedReader object as the argument in a Scanner constructor.

7. [after §16.6] Modify this chapter's WebPageReader program to read a web page through a buffer.

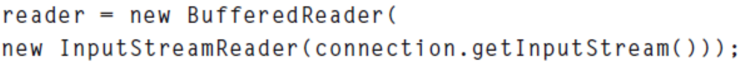
Instead of importing InputStream, import BufferedReader and InputStreamReader.

Instead of:



```
instream = connection.getInputStream();
webIn = new Scanner(instream);
```

use:



```
reader = new BufferedReader(
new InputStreamReader(connection.getInputStream()));
```

And instead of:

```
while (i < maxI && webIn.nextLine())
{
  System.out.println(webIn.nextLine());
  i++;
}
```

use:

```
for (int i=0; i < maxI; i++)
{
  System.out.println(reader.readLine());
}
```

First sample session:

```
Enter a full URL address: htp://www.whaleWatching.org
class java.net.MalformedURLException
unknown protocol: htp
```

Second sample session:

```
Enter a full URL address: https:/www.whaleWatching.org
class java.lang.IllegalArgumentException
protocol = https host = null
```

Third sample session:

```
Enter a full URL address: https://www.whaleWatching.org
Enter number of lines: 5
```

<first five source code lines from the whale watching web page>

8. [after §16.7] Describe what the following program does and explain how it does it.

```
import java.nio.DoubleBuffer;

public class Temperatures
{
  public static void main(String[] args)
  {
    double[] temps = new double[]{
      63, 61.5, 59, 58.2, 57, 57.6, 58.3, 61, 63, 65.5, 68, 72,
      76.5, 79, 82, 83.5, 81.7, 79.3, 77, 75.2, 73, 70.4, 68, 66.5};
    DoubleBuffer tempBuf = DoubleBuffer.wrap(temps);
    int capacity = tempBuf.capacity();
    DoubleBuffer tempBuf2 = DoubleBuffer.allocate(capacity);
    tempBuf.position(capacity / 2);
    while (tempBuf.position() < capacity)
    {
      tempBuf2.put(tempBuf.get());
    }
    tempBuf.rewind();
    tempBuf.limit(capacity / 2);
    while (tempBuf.position() < tempBuf.limit())
    {
      tempBuf2.put(tempBuf.get() + 2.0);
    }
    temps = tempBuf2.array();
    for(double temp : temps)
    {
      System.out.print(temp + " ");
    }
  } // end main
} // end Temperatures class
```

9. [after §16.7] Indicate the modifications you'd need to make to the code in the previous exercise if you want tempBuf2 to be a ByteBuffer instead of a Doublebuffer. Assume that tempBuf continues to be a DoubleBuffer. Keep in mind that the number of bytes needed is eight times the number of doubles. You can use ByteBuffer's asDoubleBuffer method to put double values into and get double values out of a ByteBuffer, but these operations do not change the nature of ByteBuffer's underlying array of bytes. So you will not be able to use ByteBuffer's array method to transfer its contents to the original double[] array, and the final output loop will need to get its data from the ByteBuffer object instead of the original double[] array. ByteBuffer's asDoubleBuffer method will help you do this.

10. [after §16.8] Implement a program that creates a file named Lincoln.txt that contains the string "Government of the people, by the people, for the people, shall not perish from the earth." To create the file, first instantiate a Path object with Lincoln.txt for the filename. Use the getBytes method to store the Lincoln quote as a byte array. Use ByteBuffer's wrap method to create a ByteBuffer from the byte array. Then in a try-with-resources header, open a FileChannel for writing, and call the channel's write method to write the buffer to the file. Confirm that your program works by reading the generated Lincoln.txt file with a text editor.

11. [after §16.8] Implement a program that uses a FileChannel to read from a text file. Feel free to create the text file with an editor such as Microsoft NotePad or use the Lincoln.txt file from the previous exercise. Initialize a MAX_LENGTH constant with a value larger than what's needed to enable a byte array and a ByteBuffer to hold the quotation from the previous exercise. In a try-with-resources header, open a FileChannel for reading. Call FileChannel's read method with a ByteBuffer argument and do not bother to create a persistent memory map. Call ByteBuffer's get method with a byte array argument to transfer the buffer's contents to the byte array. Then use the byte array as the argument in a String constructor call that's part of a println statement that displays the contents of the file.

12. [after §16.9] Create a text file (with an editor such as Microsoft NotePad) named TwoYearsBeforeTheMast.txt which contains this text:

> Six days shalt thou labor and do all thou art able,
>
> and on the seventh—

> holystone the decks and scrape the cable.

Implement a program that creates a directory named RichardHenryDana that resides as a subdirectory in the current directory. The program moves the text file from the current directory to the new subdirectory.

In your program, instantiate one Path object for the TwoYearsBeforeTheMast.txt file and instantiate another Path object for the RichardHenryDana subdirectory. Then use those two paths with the resolve method to create a destination Path object for where the file is supposed to go. Next, check to see if the subdirectory already exists, and if it does not, create the new subdirectory. Then attempt to move the file to the subdirectory. Use a try-catch mechanism to handle failure.

13. [after §16.10] Modify this chapter's DirectoryDescription program to print name and size information for only those files whose "glob" pattern matches a user-specified string. To do this, you'll need to prompt for and read the desired "glob" pattern. In addition to Scanner, you'll need to use the Java FileSystem and PathMatcher classes to create a matcher object like that in the constructor of this chapter's FileVisitor class. Then, you'll need to put the for-each loop's printf statement inside an if statement whose condition is something like matcher.matches(name), where name comes from path.getFileName().

14. [after §16.10] Modify this chapter's FindFiles program to restrict its search to just one directory. Do this by conditionally returning SKIP_SUBTREE at an appropriate place in the FileVisitor class.

Review Question Solutions

---

1. writer = new PrintWriter("dogs.html");

2. The `<h1>` and </h2> tags enclose the visible web page header.

3. To be file-writable and file-readable, an object must be an instance of a class that implements Serializable.

4. ObjectInputStream objectIn = new ObjectInputStream(

new FileInputStream("automobiles.data"));

5. False. The readObject method returns an Object type. You must explicitly cast the returned value to the type of your reference variable before assigning it to that variable.

6. `try (PrintWriter fileOut = new PrintWriter("TradingPartners", "UTF-16"))`

7. `try (Scanner fileIn = new Scanner(Paths.get("Suppliers", "UTF-16"))`

8. Enum StandardOpenOption is in the java.nio.file package.

9. The ... notation (an ellipsis) says you may enter zero or more arguments of the type specified just before the ellipsis.

10. When a file is large, for good performance, use a buffered file handler. For text files, use a BufferedReader or a

BufferedWriter.

11. A flip() method call makes a buffer’s limit equal its current position, and then it makes the current position equal to zero.

12. ByteBuffer’s get() method is relative. It returns the byte at the buffer’s current position and increments that position. ByteBuffer’s get(int index) method is absolute. It returns the byte at index and does not alter the buffer’s current position.

13. True. ByteBuffer’s putDouble(double value) method inserts value into the buffer starting at the current position and increments position to just after that inserted value.

14. To copy integers[3] to buffer’s current position, use:

buffer.asIntBuffer().put(integers, 3, 1);

15. To open a read-only FileChannel to an existing file called “CanadianFriends,” use this:

```
FileChannel channel = new FileChannel.open(
  Paths.get("CanadianFriends"), StandardOpenOption.READ);
```

16. To map all of channel into buffer for read-only access, use this:

buffer = channel.map(FileChannel.MapMode.RE
0, channel.size());

17. This creates a stream containing information about the parent directory:

```
DirectoryStream<Path> paths =
  Files.newDirectoryStream(Paths.get(".."))
```

It goes in a try-with-resources header.

18. A “glob” is a pattern used to see if a particular string matches a specified form. In a glob, each ? identifies a position that may contain any character, and a * identifies a contiguous sequence of positions that may contain any characters.

---

[1] It takes more than one-tenth of a second for light or a radio signal to travel to the other side of the world and back—hundreds of millions of computer cycles.

[2] This particular string is “Hamming’s Motto.” Richard W. Hamming was a founder and president of the Association of Computing Machinery.

[3] By “current directory,” we mean the default directory that the JVM chooses for the current program.

[4] The patterns specified by “glob” syntax also have other options. See Java API documentation of the getPathMatcher method in the java.nio.file.FileSystem class.

GUI Programming Basics

Objectives

- Learn how Stage and Scene work together to form the structural framework for JavaFX programs.
- Instantiate, access, and update Label controls.
- Use the FlowPane container to group components in a window.

- Instantiate, access, and update TextField controls.
- Learn the concepts behind the event-driven programming framework.
- Implement and learn how to use method references and lambda expressions.
- Understand the benefits of using method references versus lambda expressions.
- Implement property binding for simultaneous updates of a bound component.
- Perform JavaFX inline styling with the setStyle method.
- Learn what's available in terms of JavaFX CSS properties.
- Learn how to implement a CSS style sheet and tie it to a JavaFX program.

- Implement and use Button controls.
- Identify the source that fired an event and/or the type of event.

Outline

17.1 Introduction 788

17.2 SimpleWindow Program 791

17.3 Stage and Scene 794

17.4 JavaFX Components 796

17.5 Label Control 797

17.6 TextField Control 799

17.7 Greeting Program 801

17.8 Event Handling 805

17.9 Property Binding 809

17.10 JavaFX CSS 812

17.11 Scene Graph Inheritance

818

17.12 Style Sheets and Cascading 821

17.13 Button Control and FactorialButton Program 826

17.14 Distinguishing Between Multiple Events 832

17.15 Colors 834

17.16 ColorChooser Program 838

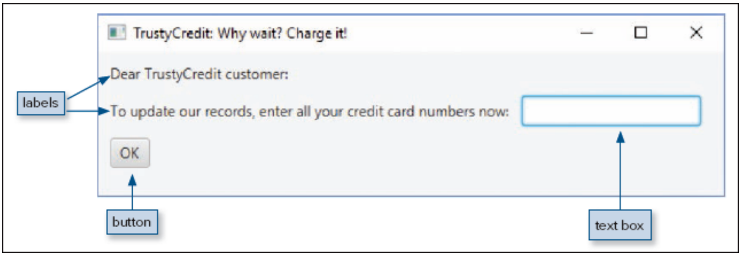
17.1 Introduction

Page 788

Hopefully, you’ve been on the edge of your seat in reading the prior chapters. If not, be prepared to be on the edge of your seat now. It’s time for the really good stuff, graphical user interface (GUI) programming.

You’ve probably heard the term GUI,

and you probably know that it's pronounced "gooey." But do GUI's three words, Graphical User Interface, make sense? "Graphical" refers to pictures, "user" refers to a person, and "interface" refers to communication. Thus, GUI programming employs pictures—like windows, labels, text boxes, buttons, and so forth—to communicate with users. For example, Figure 17.1 shows a window with two labels, a text box, and a button. We'll describe windows, labels, text boxes, and buttons in detail later on.



TrustyCredit: Why wait? Charge it!

Dear TrustyCredit customer:

To update our records, enter all your credit card numbers now:

Figure 17.1 Example window that uses two labels, a text box, and a button

©JavaFX

In the old days, program interfaces consisted of just text in a console window. Programs would prompt the user with a text question, and users would respond with a text answer. That’s what we’ve done up to now. Console-based text I/O works well in many situations, but you can’t get around the fact that some people consider text display to be boring. Many of today’s users expect programs to be livelier. They expect windows, buttons, colors, and so on for input and output. They expect GUI.

Although companies still write many text-based programs for internal use, they normally write GUI-based programs for programs that are to be used externally. It’s important that external programs are GUI based because external programs go to customers, and customers typically won’t buy programs unless they are GUI based. So if you want to write programs that people will buy, you’d better learn GUI programming.

We start this chapter with a bare-bones program where we introduce basic GUI concepts and syntax. We then describe GUI framework structures and several rudimentary GUI controls—labels, text boxes, and buttons. Generally speaking, a *control* is an object that resides within a window, but more specifically, when we refer to a "control," we're talking about an object whose class descends from the Control class in the JavaFX class hierarchy tree. For objects that reside within a window whose classes do not descend from the Control class (like shapes, images, and menus), we'll use the term *component*. The term component is generic, so it can be used for any object (including a control object) that resides within a window. With that in mind, we'll use the term component for our general discussions about window objects.

Next, we describe how users interact

with GUI programs using *event handlers*. Briefly, an event handler is a method that's executed when a user does something to the window, like click a button. We then introduce *property binding*, where two components are bound together so that a change in one component automatically generates a change in the other component. Next up is how to format windows using *Cascading Style Sheets* (*CSS*) properties.

You might have noticed the optional GUI-track sections at the ends most prior chapters?1 The GUI material in this chapter and the next two differs from that earlier GUI material in both content and method of presentation. Typically, for each end-of-chapter GUI section, we focused on one particular aspect of GUI input or output, and we employed a problem-solving approach to implement a solution for one rather specific problem. In this chapter and the next two, we present a wide range of GUI topics, and we illustrate the concepts and techniques with a sequence of relatively simple programs. These three GUI chapters will show you how to use prewritten GUI software and prescribed organizational strategies to implement a wide range of GUI presentations efficiently and elegantly.

Just as you were able to understand previous end-of-chapter GUI sections without having to read this GUI chapter, so also, you will be able to understand this chapter (and the next two) without having to read any of those previous end-of-chapter GUI sections. The two sets of material are complementary, but not interdependent. If you choose to study both—in either order—you will encounter some repetition. But that repetition will not exceed the useful repetition you regularly encounter with other material in this and other textbooks.

Historical Background—AWT and Swing Platforms

Before describing the latest form of Java's GUI, let's review some of its history. In the first Java compiler, all GUI classes were bundled into one area of the Java API library known as the Abstract Windowing Toolkit (AWT). The AWT's GUI commands generate GUI components that look slightly different in different environments. In other words, if your program instantiates an AWT button component, the button will have a Macintosh look-and-feel when the program runs on a Macintosh computer, but it will have a Windows look-and-feel when it runs on a Windows computer. That leads to portability issues. AWT programs are still portable in the sense that they're able to run on different platforms. But they run differently on different platforms. If you have a persnickety customer who demands the same appearance in all environments, AWT components will likely be unsatisfactory.

One of Java's strongest selling points was (and is) its portability. So soon after Java's initial release, Java language designers proceeded to develop a set of more portable GUI components and referred to the components collectively as Swing. To make the relationship clear between the newer Swing components and the older AWT components, they used the same component names except they prefaced the newer Swing component names with a "J." For example, the AWT has a Button component, so Swing has a JButton component.

The AWT GUI components are

known as *heavyweight components*, while the Swing GUI components are known as *lightweight components*. The AWT components are said to be heavyweight because they are built with graphics commands that are part of the computer's platform (where here, "platform" refers to the low-level instructions inherent to a particular type of computer, such as Windows or Macintosh). Being part of the computer's platform, they are too "heavy" to move to other platforms. Swing components are lightweight because they are built with Java code. Being built with Java code means they are "light" enough to "swing" from one platform to another.

The Swing platform (where here, "platform" refers to the collection of Swing packages in the Java API library) includes quite a bit more than just GUI component classes. It adds lots of functionality to AWT, but it does not replace AWT entirely. In fact, for a long time, Java applet programmers typically used AWT only, even for GUI components, and they did not use the Swing platform at all. Why? Because applets rely on browsers and, sadly, many browsers used old versions of Java, versions that did not include Swing. Now, however, applets are deprecated because of security problems.

The primary AWT packages are java.awt and java.awt.event. The primary Swing package is javax.swing (javax.swing is one of several packages that use the javax prefix). The "x" in javax stands for "extension" because the javax packages were a major extension to the core Java platform.

The JavaFX Platform

In 2008, Sun introduced a new GUI platform, JavaFX, that can be used to extend Java to *Rich Internet Applications*

(RIAs). An RIA is a web application (a program that runs within a browser either on a PC or on a portable device) that is as fully functional as typical desktop applications. Although developed with RIA in mind, JavaFX is now used extensively for standard Java programs, and that's what we'll be using it for in this chapter and the next two chapters. The "FX" in JavaFX comes from the term "special effects." JavaFX is considered a "special effects" technology, and "FX" is shorthand for "effects."

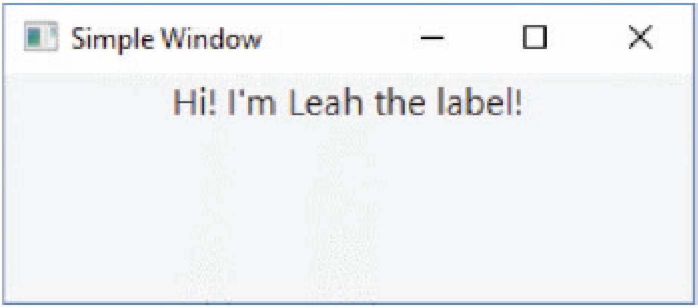
When compared to AWT and Swing, JavaFX provides a more consistent set of components, better support for touch gestures, more comprehensive formatting that matches the formatting found in web programming, better support for animation, and new special effects. In addition, JavaFX provides simpler and more efficient event handling than AWT and Swing.

In 2006, Sun Microsystems created the OpenJDK project for open-source Java development, and OpenJFX is a spinoff open-source project that supports JavaFX. Around 2014, with the advent of Java 8, Oracle (Sun's successor) went all in with JavaFX, encouraging (but not requiring) developers to make the switch from AWT/Swing to JavaFX. JavaFX is not part of Oracle's standard Java installation package, so if you want to write programs with it, you'll need to download it from OpenJFX, at https://openjfx.io.

The trend has been for new Java programs to use JavaFX, and not AWT and Swing. However, there's quite a bit of AWT and Swing code currently in production, and that means there's still a great need for Java programmers to understand the older techniques so they can update and improve existing code. As a new programmer, you're probably better off first learning JavaFX and then when you see AWT and Swing code, it shouldn't be that hard to pick it up as you go. If you'd like to learn the ins and outs of AWT and Swing, you might be able to find a copy of this book's second edition in some dusty old bookstore (remember bookstores?). Or you can peruse Chapters S17 and S18 on the book's website, which use AWT and Swing for everything GUI.

17.2 SimpleWindow Program

Later, we'll implement more useful windows that contain more than one item, but for now, let's keep things simple and examine a program that displays a single line of text inside a window, like this:



Hi! I'm Leah the label!

©JavaFX

We'll start by focusing on the most straightforward need—how to create the text item and add it to the window. To understand how to do that, you first need to be familiar with the terms JavaFX uses for its structural elements. JavaFX programs borrow from the world of theater for its terminology. Every JavaFX window is built with a Stage object. Presumably, Oracle chose the name "stage" because a stage is the foundation on which a play is presented, like a window is the foundation on which components are presented. In a play, the stage comes to life by adding one or more scenes to it. Likewise, to display something in a JavaFX window, you instantiate a

Scene object and add the Scene object to the Stage object.

In our program that displays a single line of text, we use this code:

```
Label label = new Label("Hi! I'm Leah the label!");
Scene scene = new Scene(label, WIDTH, HEIGHT);
stage.setScene(scene);
```

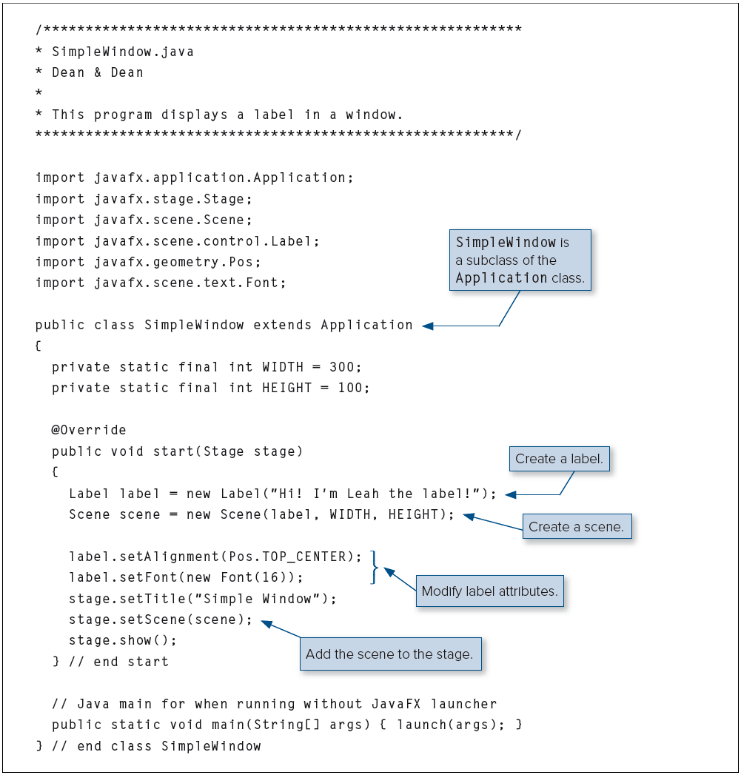
As you can see, we’re using the Label class to hold the window’s text. More specifically, we instantiate a Label object and provide the text as an argument for the Label constructor call. In this case, the text is a greeting from our personified friend, Leah the label. We then instantiate a Scene object and provide the Label control as an argument for the Scene constructor call. Next, we use the Stage object’s setScene method to add the Scene object to the stage.

The prior code is necessary and

correct, but it won't stand on its own. Now let's examine what else you need to do to support that code. The additional stuff, although tedious, will be similar for lots of programs, and, as such, you can do a lot of copying and pasting and give your brain a well-deserved break.

Figure 17.2 contains the complete SimpleWindow program that generates the window shown earlier. Notice that even this very simple program requires a relatively large number of import statements. They provide access to the Application, Stage, Scene, and Label classes. In addition, they provide access to the Pos and Font classes for layout and formatting purposes.

Note that the SimpleWindow program's class heading includes an extends Application clause. As you know, that means the SimpleWindow class is a subclass of the Application class. Application is an abstract class, and you'll use it as the superclass for all your JavaFX programs. One of the things that the Application class forces you to do is include a start method that overrides the Application class's abstract start method. This overriding start method gets a JavaFX application started like a main method gets an ordinary Java program started. Java's main method is a static method, whereas JavaFX's start method is an instance method, so there's no static modifier in the start method's heading.



```
/*************************************************************
* SimpleWindow.java
* Dean & Dean
*
* This program displays a label in a window.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.Label;
import javafx.geometry.Pos;
import javafx.scene.text.Font;

public class SimpleWindow extends Application
{
  private static final int WIDTH = 300;
  private static final int HEIGHT = 100;

  @Override
  public void start(Stage stage)
  {
    Label label = new Label("Hi! I'm Leah the label!");
    Scene scene = new Scene(label, WIDTH, HEIGHT);

    label.setAlignment(Pos.TOP_CENTER);
    label.setFont(new Font(16));
    stage.setTitle("Simple Window");
    stage.setScene(scene);
    stage.show();
  } // end start

  // Java main for when running without JavaFX launcher
  public static void main(String[] args) { launch(args); }
} // end class SimpleWindow
```

`SimpleWindow` is a subclass of the `Application` class.
Create a label.
Create a scene.
Modify label attributes.
Add the scene to the stage.

Figure 17.2 SimpleWindow program

In Figure 17.2, note the @Override annotation above the start method. As you may recall, that tells the compiler to verify that the subsequent method has the same signature as an overridden method in the superclass. In this case, because the overridden start method is an abstract method, the compiler is already required to check for matching signatures for the two start methods. Thus, for start methods, the @Override annotation doesn't provide much benefit. If you like it for its self-documentation, feel free to include it, but we'll omit it for future JavaFX programs.

As prescribed by its Application superclass, the SimpleWindow program calls its start method automatically from the background like the JRE calls an ordinary program's main method from the background. Also as prescribed by its Application superclass, the SimpleWindow program's start method receives a stage parameter that holds a reference to a Stage object. That is particularly helpful because the Stage object serves as the program's window. Thus, to modify something related to the window's display, you can use the start method's passed-in stage parameter and call its methods. For example, note these stage parameter method calls in the SimpleWindow program:

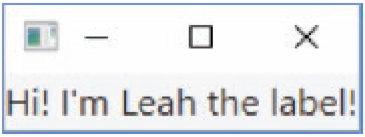
```
stage.setTitle("Simple Window");
stage.setScene(scene);
stage.show();
```

The setTitle method call adds text to the title bar at the top of the GUI window. The setScene method call adds the previously instantiated Scene object to the window. The show method call displays the scene's current contents within the window.

Near the top of the start method, you can see this statement:

```
Scene scene = new Scene(label,
WIDTH, HEIGHT);
```

It declares and initializes a Scene reference variable by calling the Scene constructor with three arguments. The first argument, label, holds a Label control, which displays the window's message. The next two arguments, WIDTH and HEIGHT are optional, and they specify the window's width and height, respectively. If you omit those arguments, then the window's size conforms to the size of its contents, like this:



Hi! I'm Leah the label!

©JavaFX

At the top of the SimpleWindow class, you can see that we use static constants for the window's width and height values:

private static final int WIDTH = 300;
private static final int HEIGHT = 100;

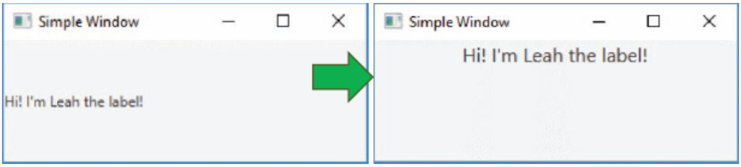
The width and height values are specified as pixels. A pixel is a computer monitor's smallest displayable unit, and it displays as a dot on the screen. If you create a scene with a width of 300 and a height of 100, your window will consist of 100 rows of pixels with 300 pixels in each row. Each pixel displays with a certain color. The pixels form a picture by having different colors for the different pixels.

To provide perspective on how large a 300-by-100 pixel window is, you need to know the dimensions, in pixels, of an entire computer screen. The dimensions of a computer screen are referred to as the screen's *resolution*. A typical resolution for a desktop monitor would be 1366 × 768 or 1920 × 1080. A typical resolution for a smartphone's display would be 720 × 1280. A 720 × 1280 resolution displays 1280 rows of pixels with 720 pixels in each row.

In the start method, note these two statements:

```
label.setAlignment(Pos.TOP_CENTER);
label.setFont(new Font(16));
```

The statements modify the label's presentation format. Specifically, they change the presentation from what's on the left below to what's on the right below:



Hi! I'm Leah the label!

Hi! I'm Leah the label!

©JavaFX

The setAlignment method call positions the label's text at the top-center of the label's area. What is the label's area? The label control expands to fill the entire window, so the label's text displays at the top of the window, centered. The top-center alignment comes from the method call's Pos.TOP_CENTER argument, where Pos is an enumerated type and TOP_CENTER is one of its values.

The setFont method call specifies the label's font with the help of a Font object argument. The Font class is a wrapper for a font's type and size, and Font's two-parameter constructor allows you to specify both of these font properties. The code above uses Font's one-parameter constructor, which tells the JVM to use the default font type and a font size of 16 points. A point is a unit of measure borrowed from the publishing industry and used by word processing software. A typical Microsoft Word document uses a default point size of 12.

At the very bottom of Figure 17.2, we have inserted a cryptic main method. Normally for a JavaFX program, including the main method is unnecessary, but we include it here to show you what you might need every now and then. If you run your JavaFX program using one of the current popular IDEs, then you don't need to include main. The IDE will generate main on its own, and the generated main method will call

Application.launch, just like in the SimpleWindow program. Likewise, if you run your JavaFX program from a command line, you don't need to include main because it gets generated automatically. If you're using an older IDE, then you'll need to include main. So to be maximally compatible, you should include main, but in the interest of saving space in the book, we will omit it starting with the next JavaFX program.

17.3 Stage and Scene

In the prior section, we used the Stage and Scene classes as part of the SimpleWindow program, but we didn't go into many details. In this section, we'll dig a little deeper. Here are API headings and descriptions for some of the more useful methods available to Stage objects:

public final void setHeight(double value)

Assigns the stage window's height in pixels.

public final void setScene(Scene value)

Specifies the stage's current scene.

public final void setTitle(String value)

Assigns the stage window's title, which appears at the top in the title bar.

public final void setWidth(String value)

Assigns the stage window's width in pixels.

public final void setX(double value)

Assigns the horizontal location of the window's top-left corner on the screen.

public final void setY(double value)

Assigns the vertical location of the window's top-left corner on the screen.

public final void show()

Displays the scene's current contents within the window.

Three of the above methods should look familiar—setTitle, setScene, and show—because you've seen them in the SimpleWindow program. Those methods are defined in the Stage class. The other methods are defined in the Window class, a superclass of the Stage class.

As described earlier, you can specify a scene's width and height with the optional second and third arguments of a Scene constructor, and the size of the JavaFX window conforms to the size of the scene. Alternatively, you can specify a window's size by calling the Stage object's setWidth and setHeight methods. As with all GUI size values (except for font sizes), those methods use units of pixels for their argument values. You can retrieve a window's size by calling the Stage object's getWidth and getHeight methods.

You can position your program's window on the computer screen by calling the Stage object's setX and setY methods with *x* and *y* argument values. The *x* and *y* arguments are integers that specify the *x*, *y* coordinate position of the window's top-left corner relative to the top-left corner of the computer screen. If *x* equals 0 and *y* equals 0, then the window's top-left corner will coincide with the computer screen's top-left corner. Positive *x* values go to the right. Positive *y* values go down. You can retrieve a window's top-left corner position by calling the Stage object's getX and getY methods.

The name "Stage" should bring to mind the idea of a stage in a theater. It's a platform upon which performers make presentations. Similarly, a JavaFX stage supports and surrounds a

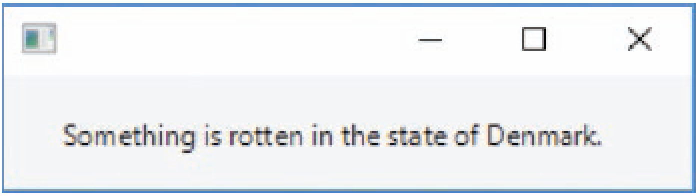
GUI program's presentation. More specifically, the Stage class forms a JavaFX window's outer frame and automatically provides the standard Windows features that you've come to know and love—a title bar, a border, a minimize button, a close-window button, the ability to resize[2] the window, and so on. This doesn't require any programming on your part. Whenever you run a JavaFX application, background API software instantiates a Stage object and makes it accessible as an argument when the Java Virtual Machine (JVM) calls your overriding implementation of Application's start method.

If you do not use Stage's setScene method to put something on the stage, the stage will be an empty window frame. Thus, to display your JavaFX program initially, you'll want to call setScene. If you'd like to update the scene's contents after the initial display, perhaps due to user interaction, you can call methods for the various components in your scene. We'll show you how to do that later on. If you want to change the scene entirely, you can call setScene again and use a separate Scene object.

The Stage object created in the background and supplied at execution need not be a program's only stage. Shakespeare's Hamlet has a play within the main play. To implement a play-within-a-play concept in a JavaFX program, we need to implement a second stage, like this:[3]

```
Stage stage2 = new Stage();
Label label = new Label(
  "\tSomething is rotten in the state of Denmark.");

stage2.setScene(new Scene(label, 300, 50));
stage2.show();
```

And here's the resulting window:

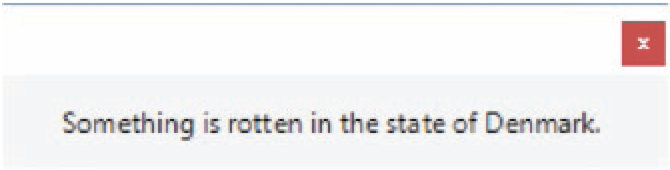


Something is rotten in the state of Denmark.

©JavaFX

If a stage.show() method call precedes the above stage2.show() method call, the secondary stage2 overlays the primary stage, but the user can drag the secondary stage's window off of the primary stage's window if he/she chooses to do so. The secondary-stage mechanism provides a handy way of warning or alerting users about exceptional situations.

By default, a secondary stage has the same header decoration and buttons as a primary stage. However, it is possible to simplify the heading of a secondary stage for a presentation like this:



Something is rotten in the state of Denmark.

©JavaFX

To obtain this simplified heading, add this import statement:

import javafx.stage.StageStyle;

Then, construct the secondary stage with a StageStyle.UTILITY argument, like this:

Stage stage2 = new Stage(StageStyle.UTILITY);

17.4 JavaFX Components

Now let's consider the objects that sit inside a window—the components. Here are some examples of JavaFX components:

- Label, TextField, Button
- TextArea, CheckBox, RadioButton, ComboBox
- ImageView
- MenuBar, Menu, MenuItem

These aren't all of the JavaFX components, just some of the more commonly used ones. We'll describe the first three components in this chapter and the other components in the next two chapters. The first two rows of component classes have the Control class as a superclass in their class hierarchy tree. The Control class supports many useful inheritable features. Along with many other methods, the Control class contains methods that handle these component features:

- foreground and background colors
- text font
- border appearance
- tool tips
- focus

For detailed information on these features, look up the Control class on JavaFX's API website.

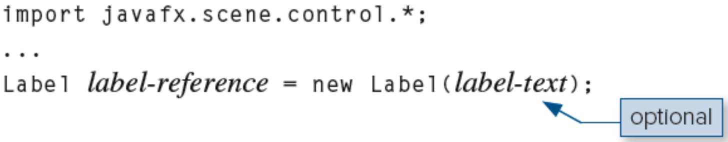
17.5 Label Control

User Interface

The Label control doesn’t do much. It simply displays text. It’s considered to be a read-only control because the user can read it but the user cannot directly change it. Normally, the Label control displays a single line of text, but that’s not always the case. You can force new lines by inserting newline characters (\n) in the Label’s text string. Or as you’ll see later, you can specify a width narrower than the text’s width and turn on text wrapping at word boundaries.

Implementation

To create a Label control, you’ll need to import the javafx.scene.control package and call the Label constructor like this:



```
import javafx.scene.control.*;
...
Label label-reference = new Label(label-text);
```

optional

The *label-text* is the text that appears in the Label control. If this argument is omitted, the Label control displays nothing. You might want to instantiate an empty label so you can fill it in later with text that's dependent on some condition.

In the SimpleWindow program, we used this statement to instantiate the Leah label:

```
Label label = new Label("Hi! I'm Leah the label!");
```

After instantiating the Leah label, we added it to the program's Scene object by including the label as an argument in a Scene constructor call. Here's the relevant code:

```
Scene scene = new Scene(label, WIDTH, HEIGHT);
```

For a real-world application, you'll normally want more than one component in a scene. In that case, you add the components to a container and add the container to the scene, with either operation first. The following statement adds a label control to a pane container (assume that pane has been instantiated as a FlowPane container; you'll learn about the FlowPane container later in the chapter):

```
pane.getChildren().add(label);
```

Note that you have to call the getChildren method to retrieve the pane container's group of contained components, and then you can add the new component (label) to the group by calling the add method.

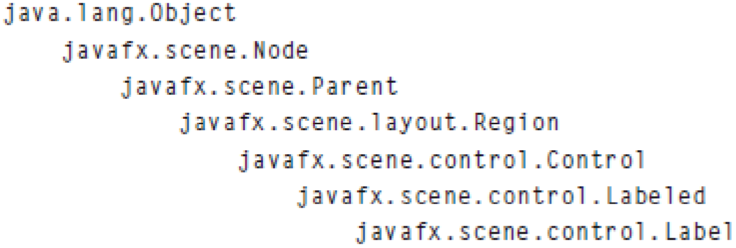
After adding components (like the above label) to the pane container's group, you can add the pane to the program's Scene object by calling the Scene constructor like this:

```
Scene scene = new Scene(pane,
WIDTH, HEIGHT);
```

Methods

As mentioned earlier, OpenJFX is the support organization for JavaFX, and it maintains an API website for JavaFX packages and classes (https://openjfx.io). OpenJFX and Oracle coordinate their release numbers, but Java and JavaFX release numbers do not need to match in a particular program. Things don't change all that much with each new JavaFX release, so what you learn in this edition of our book should coincide for the most part with what's in later JavaFX releases.

If you look up the Label class on the JavaFX API website, you'll see this class hierarchy tree:



```
java.lang.Object
    javafx.scene.Node
        javafx.scene.Parent
            javafx.scene.layout.Region
                javafx.scene.control.Control
                    javafx.scene.control.Labeled
                        javafx.scene.control.Label
```

The Label class has several methods

defined exclusively for it, but most of its methods come from its ancestor classes.

The Label class's immediate superclass is the Labeled class, which is for controls that have a label on them, like the label and button controls. The Label class inherits many methods from the Labeled class. Here are API headings and descriptions for some of the more useful ones:

public final void setAlignment(Pos alignment)

> Sets the labeled component's alignment with a Pos enumerated type argument value. In the SimpleWindow program, we used Pos.TOP_CENTER for the Pos argument.

public final void setFont(Font value)

> Sets the labeled component's alignment with a Font

argument value. In the SimpleWindow program, we used new Font(16) for the Font argument to specify a font size of 16.

public final String getText()

Returns the labeled component's text.

public final void setText(String value)

Assigns the labeled component's text. Note that the programmer can update the text even though the user cannot.

public final void setWrapText(boolean value)

If the argument is true and the labeled component's text length exceeds the width associated with the component, the text wraps to the next line when a word bumps up against the component's right edge.

As you can see further up in the Label class hierarchy tree, the Region class is an ancestor of the Label class. The Label class inherits many methods from the Region class. Here's one that we'll use in an upcoming example:

public final void setMaxWidth(double value)

Establish a maximum width in pixels for the component.

A label's default width is the width of the label's content. That's normally what you want. But imagine the situation where the label's content is populated dynamically with user input and that input is longer than the window's width. In that case, you'll want to limit the label's width (by calling setMaxWidth) and turn on line wrap (by calling setWrapText).

As you can see near the top of the Label class hierarchy tree, the Node class is an ancestor of the Label class. The Node class is the superclass for every type of object that can be added to a scene. The Label class inherits more than two hundred methods from the Node class. Here are a couple of them that we'll use later on:

public final void setStyle(String specifier)

> Sets a style feature for the Node object. The style is specified by the passed-in string argument using a JavaFX CSS property. We'll describe such properties later in this chapter.

public final void setVisible(boolean flag)

> Makes the Node object visible or invisible.

Components are visible by default, but there are some instances where you might want to call setVisible(false) and make a component disappear. For example, after you calculate a result, you might want just the result to appear without the clutter of other components. Although calling setVisible(false) makes a component invisible, it won’t cause the space to be reclaimed such that the space can be used for some new purpose. On the contrary, the invisible component’s space remains as a gap. To reclaim that space, instead of calling setVisible(false), you can call the remove method like this:

```
pane.getChildren().remove(component);
```

Or as an alternative, you can do this:

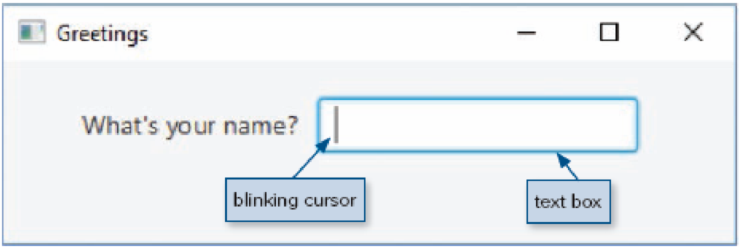
```
setVisible(false);
component.setManaged(false);
```

By doing that, you’re telling the component’s container to keep the component in memory, but do not display it when the container does its layout calculations.

17.6 TextField Control

User Interface

The TextField control (often called a text box) displays a rectangle and allows the user to enter text into that rectangle. Here's an example:


Greetings
What's your name?
blinking cursor
text box

©JavaFX

TextField Implementation

Same as the Label class, the TextField class is in the javafx.scene.control package, so you'll need to import that package. To create a TextField object, call the TextField constructor like this:

```
TextField TextField-reference = new TextField(initial-text);
```

optional

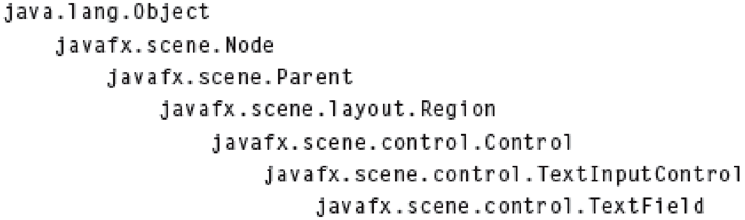
The *initial-text* is the text that appears in the text box initially. If the initial-text argument is omitted, then the text box is displayed with nothing inside it. Normally, you’ll omit the initial-text argument, but every now and then, you might want to encourage a certain user input. For example, for a customer feedback text box, you might want to populate it with something like “Best service I’ve ever received!” If the initial-text argument is included, the user can choose to enter text that replaces the initial-text string or do nothing in which case the program will use the initial-text string as the user’s input. If the user enters more characters than the text box can display at one time, the leftmost characters scroll off the display.

Same as with Label controls, to add a TextField control to a container named pane, use this syntax:

pane.getChildren().add(*TextField-reference*);

TextField Methods

The TextField class hierarchy tree is similar to the Label class hierarchy tree in that they both have Control as an ancestor class. Here's the TextField class hierarchy tree:



```
java.lang.Object
    javafx.scene.Node
        javafx.scene.Parent
            javafx.scene.layout.Region
                javafx.scene.control.Control
                    javafx.scene.control.TextInputControl
                        javafx.scene.control.TextField
```

As you can see, the only difference from the Label class hierarchy tree is that the TextField class's immediate superclass is the TextInputControl class instead of the Labeled class. The TextField class inherits many useful methods from the TextInputControl class. For example:

public void appendText(String text)

Appends the specified text to the end of the text box's current contents.

public final int getLength()

Returns the number of characters in the text box's contents.

public final String getText()

Returns the text box's contents.

public final void setEditable(boolean flag)

Makes the text box editable or non-editable.

public final void setPromptText(String text)

Inserts grayed-out text into the text box to instruct user on what to enter. The text is not used as user input. If the user enters something, the prompt text gets overlaid with the entered text.

public final void setText(String text)

Assigns the text box's contents.

Text boxes are editable by default, which means that users can type inside them. If you want to prevent users from editing a text box, call setEditable with an argument value of false. Calling setEditable(false) prevents users from updating a text box, but it does not prevent programmers from updating a text box. Programmers can call the setText method regardless of whether the text box is editable or noneditable.

Here are some of the more popular methods defined in the TextField class itself:

public final void setPrefColumnCount(int value)

Specifies the width of the text box in number of text columns.

public void setOnAction(EventHandler<ActionEvent> handler)

Registers an event handler with the text box.

The setPrefColumnCount's argument is supposed to be roughly the number of maximally wide characters (e.g., W's) that can fit in the box, but the box is usually wider than that. Regardless of the preferred column count value, the user can enter as many characters as he/she wants. If there are more characters than can fit in the box, the leftmost characters scroll out of view.

When a TextField control calls setOnAction, the JVM registers an event handler with the text box. That enables the program to respond to the user pressing Enter when the cursor is in the text box. The next section's program has code that does this.

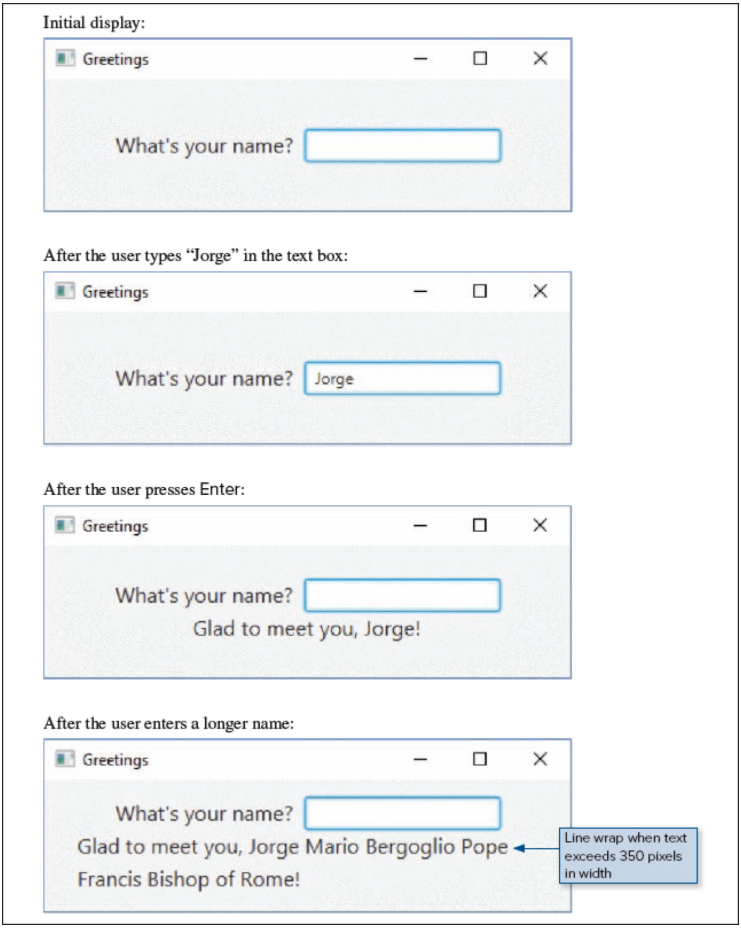
17.7 Greeting Program

Now let's examine a program that incorporates what we've learned in the past several sections, using both label and text box controls. See Figure 17.3, which shows a sample session for the Greeting program. The program prompts the user to enter a name and then responds with a personal greeting.

The Greeting program uses a FlowPane container to group together its three components—a label for the prompt, a text box for the user input, and another label for the message at the bottom. The "flow" in FlowPane comes from the manner in which components get positioned as they're added to the pane. By default, the first component gets positioned in the top-left corner. Subsequent components get positioned at the right of the prior components using a left-to-right, top-to-bottom flow. If a new component bumps into the container's right edge, it flows to the next row. As you'll see later, the Greeting program overrides the default left-side alignment and uses center alignment instead. But either way, the components are added with the aforementioned left-to-right, top-to-bottom flow. So in Figure 17.3, note how the message label flows down to the second row because there is no room for it on the first row.

Whenever you have more than one visible element in a window (such as the three components in the Greeting program), those elements must be Node objects that are part of a *scene graph*. A scene graph is a hierarchical tree of Node objects where the node at the top of the tree is considered to be the *scene graph's root*. The Node objects are controls (such as buttons and text boxes) and shapes (such as rectangles and circles), plus containers that hold the other Node objects.

For the Greeting program, the root node is a FlowPane container object, as you'll see when we examine the program's Java source code. If you look up FlowPane in the Java API, you'll see that FlowPane, as well as the Label and TextField controls in the Greeting program, are all subclasses of the Node class.



Initial display:

What's your name?

After the user types “Jorge” in the text box:

What's your name? Jorge

After the user presses Enter:

What's your name?

Glad to meet you, Jorge!

After the user enters a longer name:

What's your name?

Glad to meet you, Jorge Mario Bergoglio Pope
Francis Bishop of Rome!

Line wrap when text exceeds 350 pixels in width

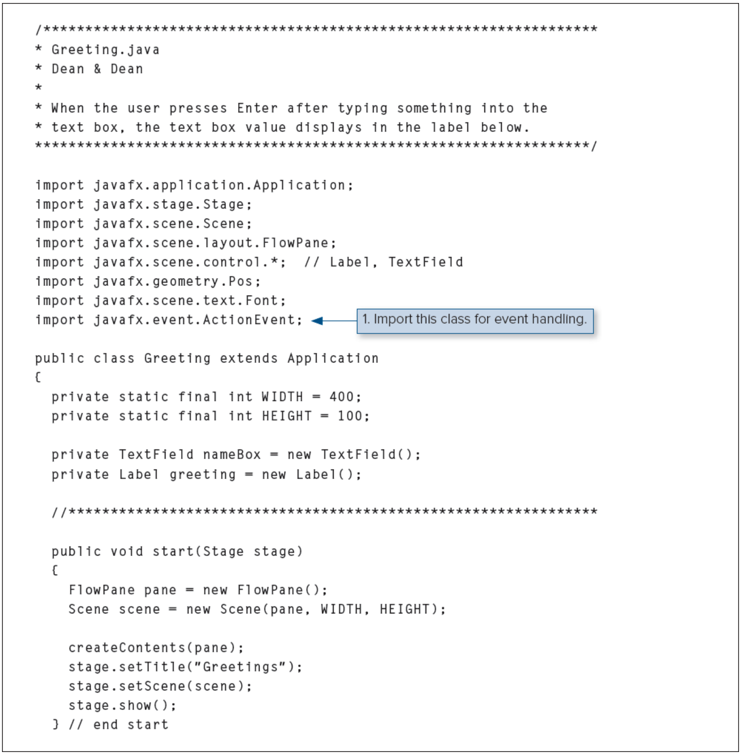
Figure 17.3 Greeting program’s sample session

©JavaFX

Page 803

Skim through the Greeting program’s source code in Figures 17.4a and 17.4b. Most of the code in Figure 17.4a should look familiar because it closely parallels the code in the SimpleWindow program. One significant difference is that we use the FlowPane class, which serves as the root node container for the program's scene graph. Here's the code that instantiates the FlowPane object and establishes it as the scene's root node:

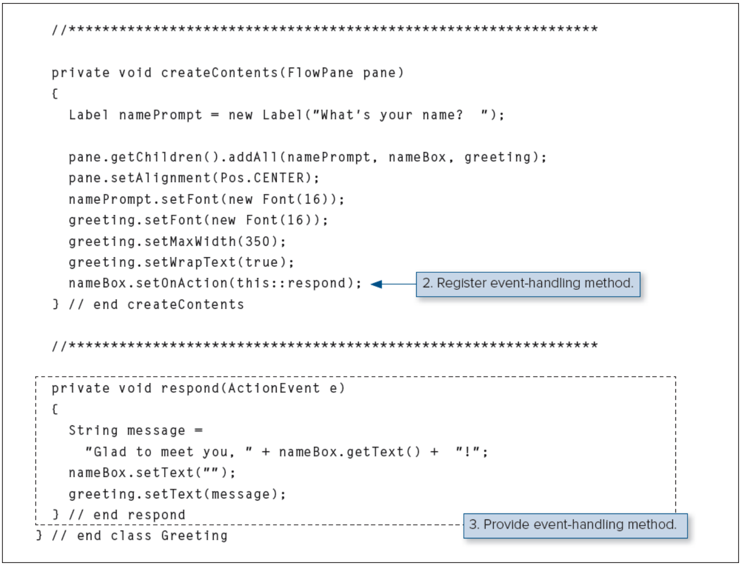
```
FlowPane pane = new FlowPane();
Scene scene = new Scene(pane, WIDTH, HEIGHT);
```



```
/*******************************************************************
* Greeting.java
* Dean & Dean
*
* When the user presses Enter after typing something into the
* text box, the text box value displays in the label below.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*;  // Label, TextField
import javafx.geometry.Pos;
import javafx.scene.text.Font;
import javafx.event.ActionEvent;

public class Greeting extends Application
{
  private static final int WIDTH = 400;
  private static final int HEIGHT = 100;

  private TextField nameBox = new TextField();
  private Label greeting = new Label();

  //****************************************************************

  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane, WIDTH, HEIGHT);

    createContents(pane);
    stage.setTitle("Greetings");
    stage.setScene(scene);
    stage.show();
  } // end start
```

1. Import this class for event handling.

Figure 17.4a Greeting program—part A



```
  //*******************************************************************

  private void createContents(FlowPane pane)
  {
    Label namePrompt = new Label("What's your name?  ");

    pane.getChildren().addAll(namePrompt, nameBox, greeting);
    pane.setAlignment(Pos.CENTER);
    namePrompt.setFont(new Font(16));
    greeting.setFont(new Font(16));
    greeting.setMaxWidth(350);
    greeting.setWrapText(true);
    nameBox.setOnAction(this::respond);
  } // end createContents

  //*******************************************************************

  private void respond(ActionEvent e)
  {
    String message =
      "Glad to meet you, " + nameBox.getText() + "!";
    nameBox.setText("");
    greeting.setText(message);
  } // end respond
} // end class Greeting
```

2. Register event-handling method.

3. Provide event-handling method.

Figure 17.4b Greeting program—part B

Page 804

The second import statement, by using the * wildcard, handles the imports for both controls—label and text box. The third import statement is for the ActionEvent class, which is needed for event handling. We'll describe event handling in detail in the next section.

In the interest of modularization, the Greeting program's start method calls a helper method, createContents, to create the components that go inside the pane. As you can see in Figure 17.4b, the createContents method contains nine lines (including one blank line). With only nine lines and there being only one call to the createContents method, there's no urgent need for a helper method, but we want you to form good habits. For this example, there are only three components and one event handler. But most GUI programs have more than three components and multiple event handlers, and for that, quite a lot of code is needed. If you stick all that code in the start method, the start method would be inordinately long. Better to break things up and stick that code in a helper method so the code is easier to understand.

In the Greeting program, you can see that we've declared and initialized one component reference variable, namePrompt, in the createContents method. In general, it's better to keep variables as local as possible, and that's why namePrompt is declared as a local variable. But if you go back to the Greeting program's instance variable section in Figure 17.4a, you can see two component reference variables, nameBox and greeting, declared there. So why use instance variables for nameBox and greeting rather than declare them locally in the createContents method? Because we need to access those components not only in createContents, but also in the event handler method (which we'll get to shortly), and that's easier when the components are instance variables. With that said, you should be aware that it would be possible to declare nameBox as a local variable in the createContents method and retrieve it from the event handler method by calling a getSource method; we'll defer that added complexity until later in the chapter.

In Figure 17.4b's createContents method, you can see that we add three components to the window. Specifically, we use a FlowPane container named pane to call getChildren().addAll() like this:

pane.getChildren().addAll(namePrompt, nameBox, greeting);

Previously, we used the add method to add one component to a FlowPane container's group of components; this time, we're using the addAll method because we're adding multiple components.

In the createContents method, after calling addAll, we call setAlignment and setFont to adjust the layout and text font (as you saw with the SimpleWindow program). Next we call greeting.setMaxWidth(350) and greeting.setWrapText(true), so that if the greeting label's contents exceed the threshold of 350 pixels, the JVM will wrap the label's text to the next line. Go back to Figure 17.3 and note the line wrap that occurs in the bottom screenshot for the Pope's eight-word name.

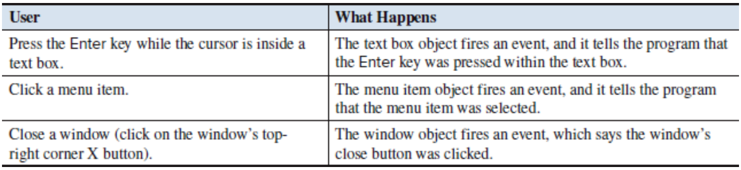
The last statement in the createContents method, nameBox.setOnAction(this::respond), registers an event-handling method with the namebox component. That statement probably looks rather mysterious. Don't worry, we'll explain it in detail in the next section. For now, just realize that it causes the JVM to call the respond event-handling method whenever the user presses the Enter key. Take a look at the respond event-handling method in Figure 17.4b. This method retrieves the user-entered text in the nameBox text box, resets namebox's contents to the empty string, and inserts a message (with the retrieved user entry) into the greeting label.

17.8 Event Handling

Most GUI programs use event-driven programming. The basic idea behind event-driven programming is that the program waits for events to occur and the program responds to events if and when they occur.

Terminology

So what is an event? An event is an object that tells your program that something has happened. For example, if the user clicks a button, then an event is generated, and it tells your program that a particular button was clicked. More formally, when the user clicks a button, we say that the button object fires an event. Note these additional event examples:



User	What Happens
Press the Enter key while the cursor is inside a text box.	The text box object fires an event, and it tells the program that the Enter key was pressed within the text box.
Click a menu item.	The menu item object fires an event, and it tells the program that the menu item was selected.
Close a window (click on the window's top-right corner X button).	The window object fires an event, which says the window's close button was clicked.

If an event is fired, and you want your program to handle it, you need to create an *event handler* (or "handler" for short) for the event. In addition, you need to associate that event with particular GUI object(s). For example, if you want your program to do something when the user clicks a particular button, you need to create a handler for the button and register that handler with the button. An event handler is sometimes referred to as a *listener*. That makes sense if you think of a handler as a dutiful attendant who listens carefully with a big ear. If an event is fired and there's nothing listening to it, that event is never "heard" and there's no response to it. On the other hand, if there is a handler listening to a fired event, the handler "hears" the event and responds by performing a specified set of operations. Figure 17.5 depicts a button being pressed (see the mouse pointer), an event being fired (see the sound waves), a handler hearing the event (see the ear), and the handler's code being executed (see the arrow going down the event-handler code). This system of event handling is known as the *event-delegation model*—event handling is "delegated" to a particular handler.

Registering an event handler is like registering your car. When you register your car, nothing much happens at that point. But later, when some event occurs, your car registration comes into play. What event would cause your car registration to be used? If you get caught speeding, the police can use your registration number as part of a traffic citation. If you get into a wreck, your insurance company can use your registration number to raise your insurance rates.

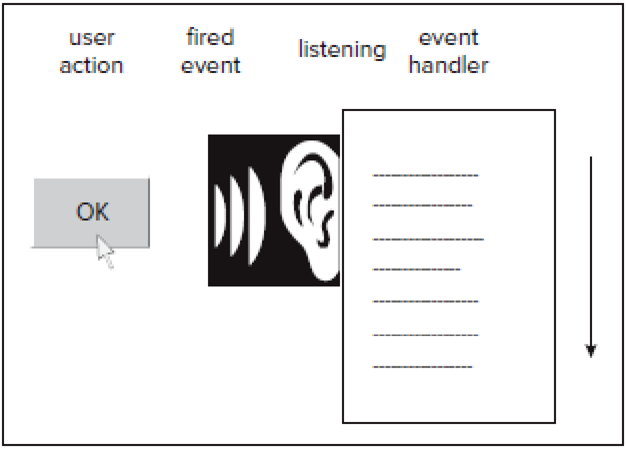

user action
fired event
listening
event handler
OK

Figure 17.5 What happens when a button is pressed

Event-Driven Programming Framework

Based on the above description, event-driven programming may feel like an altogether new type of programming, particularly the part about firing an event and listening for a fired event. Many people are fine with the idea of event-driven programming being a new type of programming. But really it's just object-oriented programming with window dressing—make that lots of window dressing. JavaFX provides an extensive collection of GUI classes that, together, form a framework on which to build GUI applications. That framework is comprised of classes, methods, inheritance, and so on. In other words, it's comprised of OOP components. As a programmer, you don't have to understand all the details of how the framework works; you just have to understand it well enough to use it. For example, you have to know how to plug in your event handlers properly. Figure 17.6 provides a high-level, graphic illustration of what we're talking about.

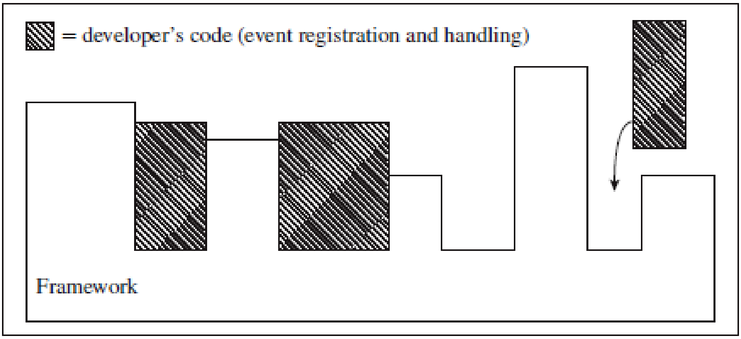


Figure 17.6 Event-driven programming

framework

Why did the Java language designers bother to provide the event-driven programming framework? It satisfies the goal of getting maximum benefit from minimum code. With the help of the framework, Java programmers can get a GUI program up and running with a relatively small amount of effort. Initially, the effort might not seem so small, but when you consider all that the GUI program does (automatic event firing, listening for fired events, and so on), you'll find that your return on investment is quite good. Here are the steps needed to implement a handler for a GUI component like the nameBox in the earlier Greeting program. These steps correspond to the numbered callouts in Figures 17.4a and 17.4b:

1. For the standard type of event handling used in the earlier

Greeting program, import the ActionEvent class from the javafx.event package. In Figure 17.4a, you can see that the ActionEvent class serves as the type for the respond event handler's parameter. For the event handler to work, you're required to have an explicit ActionEvent parameter, even if the event-handler method does not use it.

2. Register the event-handling method. In Figure 17.4b our Greeting program does this by having the text box component call the setOnAction method with a *method reference* argument. The method reference argument ties the event handler method to the component that's been fired. We'll describe method references in the next subsection. Another way to register an event-handling method is to have the registering method's argument be a *lambda expression*, which itself includes all the code needed to handle the event. We'll describe this alternative after our description of method references.

3. Supply the event-handling code. This could be in a separate method-reference method like that in Figure 17.4b, or it could be in a lambda expression.

Method References

As shown in the Greeting program in Figure 17.4b, to implement an event handler using a method reference, we create a helper method with this heading:

```
private void respond(ActionEvent e)
{
...
```

The method name can be anything, but the return type must be void, and there must be exactly one parameter of type

ActionEvent. We position this event-handling method with the other helper methods—typically near the bottom of your program's code. In the Greeting program, we register the respond event-handling method with the nameBox component. Here's the registration code:

```
nameBox.setOnAction(this::respond);
```

The special :: notation for method references is how the compiler ties the method (respond in this case) to the class where the method is defined. In most event handling situations, you'll want to define the event handler in the same class as the class that contains the setOnAction method call. To indicate the same class, we use our old friend this, which refers to the calling object (and its class). If you want to define the event-handler method in a different class, then make the method a static method and for the setOnAction method call's method-reference argument, preface the :: with the other class's class name.

Method-reference event handling is versatile because it's modular. You can register the same event-handling method with more than one component by having the different components each call the setOnAction method.

Lambda Expressions

Using a method reference for event handling works great in terms of functionality, but the technique can be less than ideal in terms of code length. A lambda expression4 provides the same functionality, but with less code. A lambda expression's primary code reduction comes from not requiring a method name or a method heading. Here's how the Greeting program's event handling can be rewritten with a lambda expression instead of a method reference:

```
nameBox.setOnAction((ActionEvent e) -> {
  String message =
    "Glad to meet you, " + nameBox.getText() + "!";
  nameBox.setText("");
  greeting.setText(message);
});
```

Notice that there's no respond method; only the parameter from the method, an arrow (->), and the method's body. Pretty cool, eh? But wait, there's more. You can omit the parameter's type because the compiler can infer its type by examining the calling method's signature. In this case, the calling method is setOnAction and its signature has one parameter. That parameter is a method with an ActionEvent parameter. Thus, you can omit ActionEvent in the prior lambda expression. Here's the resulting streamlined code:

```
nameBox.setOnAction(e -> {
```

Note that in addition to eliminating the parameter's type, we also eliminated the parameter's enclosing parentheses. You can eliminate a lambda expression's parameter parentheses only if there is just one word; that means only one parameter and there's no type for that parameter.

⚠ As a further code-reducing measure, you can eliminate the braces around a lambda expression's body if the body consists of just one statement. And for that single statement, you can eliminate its closing semicolon. Feel free to do that, but be aware that if someone wants to add a second statement later on, they'll have to remember to add semicolons after each statement and add surrounding braces.

By the way, that `->` thing that goes between the lambda expression's parameter and its body—that's called the *arrow operator* or *arrow token*.

In the lambda expression code fragment shown earlier, note that the lambda expression does not use its ActionEvent e parameter. Even though e is not used, you’re still required to provide it at the top of the lambda expression. The lambda expression serves as an argument for the setOnAction method call, and the setOnAction method’s signature specifies an event-handler parameter where the event handler uses an ActionEvent parameter. Thus, if you call setOnAction, you are required to provide an argument that is an event-handler method, and that method must have an ActionEvent parameter.

So are you now sold on lambda expressions and wondering why anyone would ever bother with method references? As mentioned earlier, method references can be easily reused and called by different components. Because a lambda expression has no method name, for it to be reused, you have to duplicate its code wherever you want to use it. Duplicate/redundant code makes maintenance chores more difficult. So for upcoming examples, we'll use method references for cases where we might want to reuse an event handler. When there's no need to reuse, we'll use lambda expressions for the most part, but that's not a hard and fast rule.

17.9 Property Binding

In the Greeting program, with both the method-reference technique and also with the lambda expression technique, we implemented the event-handling functionality by having the nameBox component call the setOnAction method. The "action" that the setOnAction method looks for is the user pressing Enter in the text box. As an alternative, let's get fancy and implement an event handler that responds to individual keystrokes rather than waiting for the user to press Enter. The easiest way to do that is to bind the greeting label's text property to the nameBox's text property.

Property binding is a relatively new technique introduced with JavaFX. It's where a property of a target object is bound to that same type of property in a source object. So when the source object's property changes, the target object's property changes simultaneously.

The syntax for property binding is pretty straightforward. Just call the bind method like this:

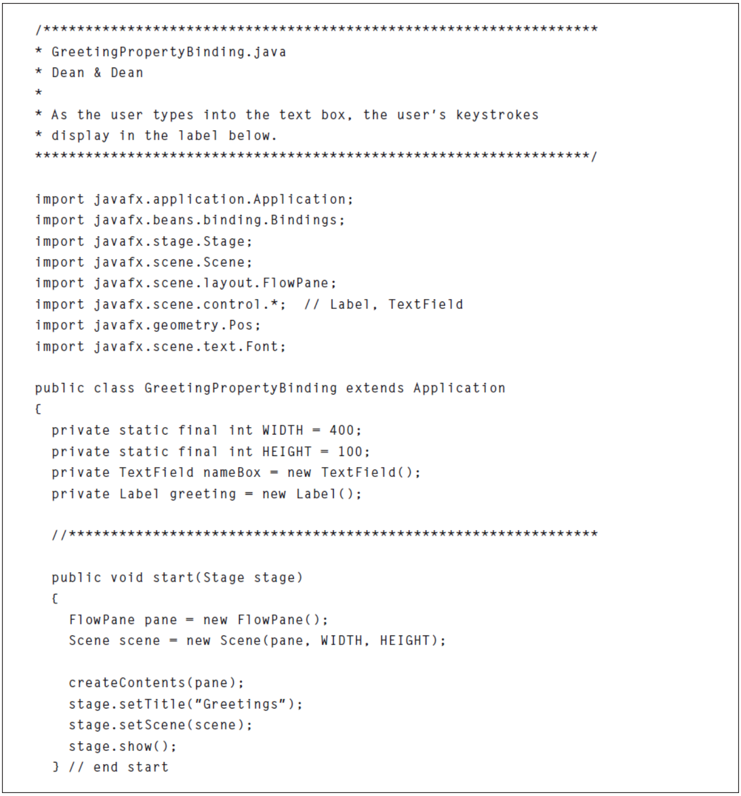
target-object-reference.text-property.bind(*source-object-reference.text-property*);

For the next version of the Greeting program, we use property binding and bind the greeting label's text property to the nameBox's text property. Here's the code for how that binding can be done:

```
greeting.textProperty().bind(nameBox.text
```

Note that greeting is the reference variable for the target object, and nameBox is the reference variable for the source object. So as the user enters text into the namebox text box, the greeting label displays that text.

Figures 17.7a and 17.7b show the GreetingPropertyBinding program. Most of its code is identical to what's in the original Greeting program, but it's streamlined considerably. It omits the ActionEvent import statement at the top, it omits the call to setOnAction in the createContents method, and it omits the respond method in its entirety. With property binding, there's no longer a need to register an event handler with the text box. Thus, there's no longer a need to call the setOnAction method for the registration, and there's no longer a need to include the respond event-handler method. The respond method uses the ActionEvent class as a parameter type in its heading, and without the respond method, the ActionEvent import statement becomes unnecessary.

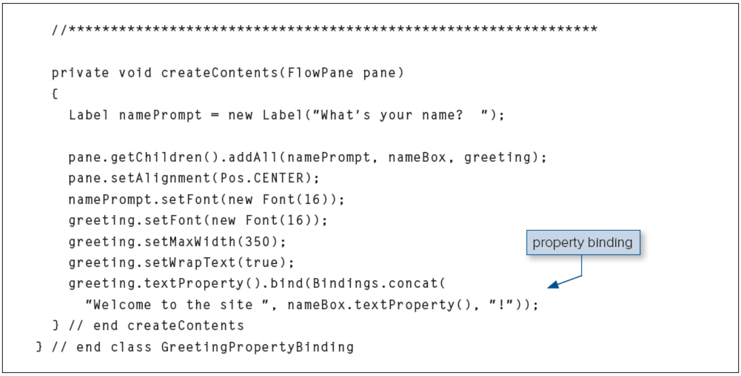


```
/*******************************************************************
* GreetingPropertyBinding.java
* Dean & Dean
*
* As the user types into the text box, the user's keystrokes
* display in the label below.
*******************************************************************/

import javafx.application.Application;
import javafx.beans.binding.Bindings;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*;  // Label, TextField
import javafx.geometry.Pos;
import javafx.scene.text.Font;

public class GreetingPropertyBinding extends Application
{
  private static final int WIDTH = 400;
  private static final int HEIGHT = 100;
  private TextField nameBox = new TextField();
  private Label greeting = new Label();

  //*****************************************************************

  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane, WIDTH, HEIGHT);

    createContents(pane);
    stage.setTitle("Greetings");
    stage.setScene(scene);
    stage.show();
  } // end start
```

Figure 17.7a GreetingPropertyBinding program—part A

In the GreetingPropertyBinding program's createContents method, note the code that's in charge of the property binding. It's a bit more complicated than the code presented earlier. As you can see below, we bind the greeting label not only to the nameBox text box, but also some surrounding text:

greeting.textProperty().bind(Bindings.conc
"Welcome to the site ", nameBox.textProperty(), "!"));



```
  //*****************************************************************

  private void createContents(FlowPane pane)
  {
    Label namePrompt = new Label("What's your name?  ");

    pane.getChildren().addAll(namePrompt, nameBox, greeting);
    pane.setAlignment(Pos.CENTER);
    namePrompt.setFont(new Font(16));
    greeting.setFont(new Font(16));
    greeting.setMaxWidth(350);
    greeting.setWrapText(true);
    greeting.textProperty().bind(Bindings.concat(
      "Welcome to the site ", nameBox.textProperty(), "!"));
  } // end createContents
} // end class GreetingPropertyBinding
```

property binding

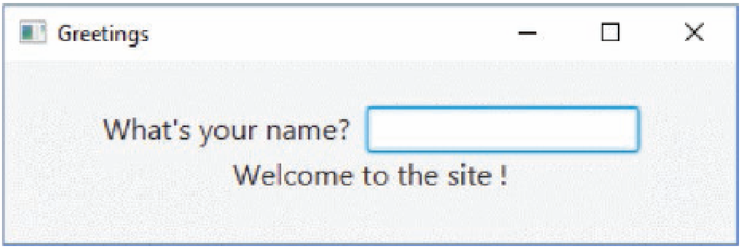
Figure 17.7b GreetingPropertyBinding program—part B

Page 811

If you look up the bind method's API heading, you'll see that its parameter is defined to be an Observable interface. The textProperty method returns a StringProperty, which implements the Observable interface, so we could use nameBox.textProperty() by itself for the bind method call's argument. But we want the greeting label to be more expressive than simply repeating the user's name. We want the name to be part of a "Welcome to…" message. Thus, we enlist the help of the Bindings.concat method to concatenate the welcome message and the user's name. The concat method returns a StringExpression, which also implements the Observable interface, so it is allowed as an argument for the bind method call.

By binding the greeting label to the user-input-embedded welcome message, when the user presses a key in the nameBox text box, the JVM immediately displays that key in the welcome message. Here's what we're talking about:

Initial display:

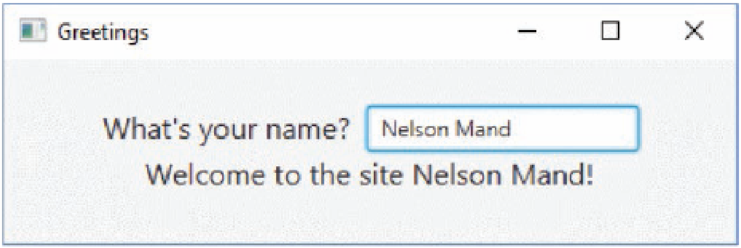


What's your name?

Welcome to the site !

©JavaFX

After the user types “Nelson Mand” in the text box (without pressing Enter):



What's your name? Nelson Mand

Welcome to the site Nelson Mand!

©JavaFX

Note that when the window first loads and before the user does anything, the greeting label displays the welcome message with the empty string for the user’s name.

17.10 JavaFX CSS

When you use JavaFX to adjust the formatting for something in the program's window, that's called *styling* the window. In the preceding SimpleWindow and Greeting programs, we *styled* the label controls' font sizes by calling setFont(new Font(16)). Assuming you have a label control named label, if you'd like to adjust the label's text color, the size of the invisible box that surrounds the label, and the label's position within the box, you could do this:

```
import javafx.scene.paint.Color;
import javafx.geometry.Pos;
...
label.setTextFill(Color.BLUE);
label.setPrefWidth(400);
label.setPrefHeight(100);
label.setAlignment(Pos.CENTER);
```

Using multiple method calls, like setTextFill and setPrefWidth can be handy in certain circumstances, but the trend is to use JavaFX CSS properties instead. CSS stands for Cascading Style Sheets. CSS is one of the foundational building blocks of web programming along with HyperText Markup Language (HTML) and JavaScript.

With CSS's ubiquitous presence on the web, Oracle decided to jump on the bandwagon and has pushed its version of CSS (JavaFX CSS) as one of the key features of JavaFX. With JavaFX CSS using the same basic syntax and properties as standard CSS, it makes learning JavaFX easier for web programmers. JavaFX CSS provides a wide array of properties that enable programmers to finely tune how their GUI programs are formatted. Borrowing from the web page CSS model, JavaFX CSS properties can be applied to containers within a window’s scene graph, and those properties can flow down to the nodes that the container contains. This feature furthers the goals of consistency and software reuse.

One way to apply JavaFX CSS properties to your program is to put the properties in an external file, called a *style sheet*, and connect that file to your program. We’ll discuss that technique in an upcoming section. As an alternative, you can use an *inline style*. That’s when you call the setStyle method and provide the JavaFX CSS properties as a string argument in the method call. For example, here’s how you could use setStyle to adjust the label control in terms of font size, the label’s text color, the size of the invisible box that surrounds the label, and the label’s position within the box:

```
label.setStyle(
    "-fx-font-size: 28; " +
    "-fx-text-fill: blue; " +
    "-fx-pref-width: 400; " +
    "-fx-pref-height: 100; " +
    "-fx-alignment: center;");
```

The setStyle method’s string argument consists of a list of property-value pairs, where each property-value pair is followed by a semicolon. So in the above code fragment, you can see there are five property-value pairs with the first property-value pair being "-fx-font-size: 28; ". Note that each property-value pair uses a colon to separate the property from its value. Also note that each property starts with the prefix -fx-. After the -fx- comes a descriptor that roughly matches one of the properties from the standard CSS syntax.

In the prior code fragment, the blue and center property values are self-explanatory. The 28 for font size indicates that the label is to be displayed with a 28-point font size. A point is a unit of measure used extensively with word processing software. A typical Microsoft Word document uses a 12-point font size. The 400 and 100 for the label's preferred width and preferred height properties indicate that the label's surrounding box is to be displayed with dimensions of 400 × 100 pixels.

The setStyle method is defined in the Node class, which is high up in the JavaFX class hierarchy. Thus, all JavaFX components are able to call the setStyle method. Be aware that if the same object calls ⚠ the setStyle method twice, all the property-value pairs that were set with the first call are wiped out, regardless of whether the second call's properties overlap the first call's properties. For example, suppose you include both of these setStyle method calls in your program:

```
label.setStyle("-fx-alignment: center;");
label.setStyle("-fx-font-size: 16;");
```

The alignment setting would be lost, and the label would not be centered.

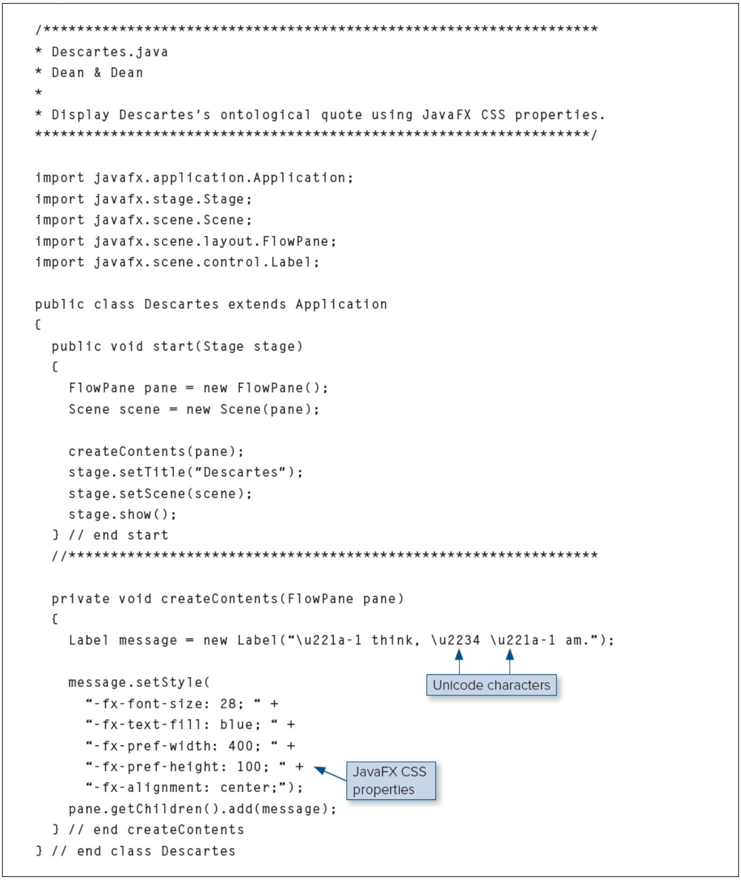
If you need to come back later and add another style specification to a previously styled component, you can’t just call setStyle again with another style specification, because that would erase the original style specification. The solution is to extend the previous property-value pair list. More specifically, you should retrieve the current list of property-value pairs and concatenate the new property-value pair(s) to it, like this:

```
label.setStyle(label.getStyle() + " -fx-font-size: 16;");
```

Kind of ugly, but it gets the job done.

Example Program

The Descartes program in Figure 17.8a uses JavaFX CSS properties to display French philosopher René Descartes’s famous quote “I think, therefore I am.” Note the setStyle method call. It uses the code shown earlier that formats the label’s font size, color, surrounding box size, and position. Note how that formatting code is manifested in the resulting window.



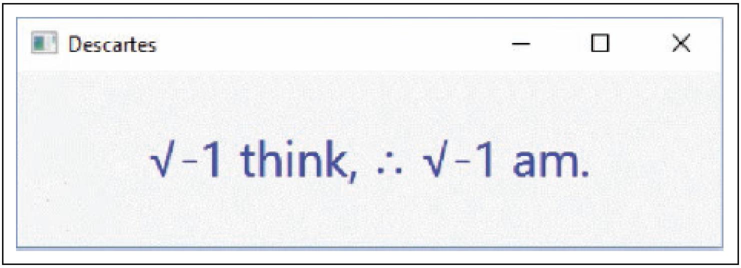
```
/*************************************************************************
* Descartes.java
* Dean & Dean
*
* Display Descartes's ontological quote using JavaFX CSS properties.
*************************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Label;

public class Descartes extends Application
{
  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane);

    createContents(pane);
    stage.setTitle(”Descartes”);
    stage.setScene(scene);
    stage.show();
  } // end start
  //*********************************************************************

  private void createContents(FlowPane pane)
  {
    Label message = new Label(“\u221a-1 think, \u2234 \u221a-1 am.”);

    message.setStyle(
      “-fx-font-size: 28; “ +
      “-fx-text-fill: blue; “ +
      “-fx-pref-width: 400; “ +
      “-fx-pref-height: 100; “ +
      “-fx-alignment: center;”);
    pane.getChildren().add(message);
  } // end createContents
} // end class Descartes
```

Figure 17.8a Descartes program

The Descartes program instantiates its Scene object with just one argument, pane, like this:

Scene scene = new Scene(pane);

You might recall that with no width and height arguments in the Scene constructor call, the window's size conforms to the size of its contents. Therefore, when the message label's dimensions are assigned with the -fx-pref-width: 400; and -fx-pref-height: 100; property-value pairs, that determines the window's size.



```
√-1 think, ∴ √-1 am.
```

Figure 17.8b Descartes program output

©JavaFX

The Descartes program uses Unicode character symbols to represent the words *therefore* and *I*. The Unicode character \u2234 displays ∴, which is the logic symbol for "therefore." The Unicode character \u221a displays √, which is the left side of a square root symbol. In the resulting window, note how the square root symbols are at the left of −1's. Can you figure out why √ −1 represents I? Because the square root of −1 is the imaginary number i. Please pardon the attempt to entertain with a mathematical pun; we're math nerds. Figure 17.8b shows the program's output.

Popular Properties, Their Allowed Values, and Classes Defining Those Properties

You can find a comprehensive presentation of JavaFX CSS properties and related details at https://openjfx.io/javadoc/13/javafx.graphics/javafx/scene/doc-files/cssref.html. There are lots of properties and lots of details, and unless you use the properties on a regular basis, you're not going to be able to memorize them all. Just try to know what's available, so you can go back and look up the details of a particular property when there's a need. To help in that regard, the table in Figures 17.9a and 17.9b summarizes the more important JavaFX CSS properties, their corresponding allowed values, and the JavaFX classes that those properties can apply to. Some of the classes should look familiar from earlier in the chapter, and you'll see some of the other ones later on. Still other ones, we don't mention; if you're curious about them, feel free to look them up online.

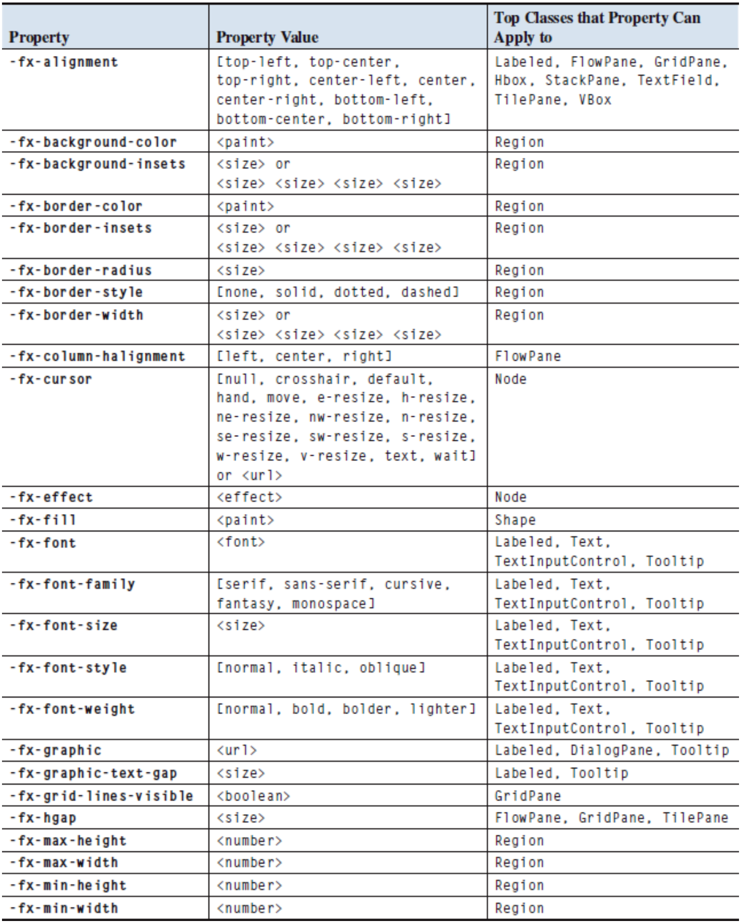
In the table's left column, you can see that almost every property begins with -fx-. The first word after the -fx- prefix identifies a principal property. When a principal property has a subordinate property, a dash is appended to the principle property, followed by the subordinate property. For example, -fx-border-color is comprised of border, a principle property, and color, a subordinate property.

In the table's middle column, you can see that the property values are surrounded by either [ ]'s or < >'s. The [ ]'s surround values that are to be used as is. The < >'s surround property value types. We'll describe those types in the next subsection.

The table's right column shows the top classes that the property can apply to. We say "top" classes because there are additional classes that the property can apply to. All of the top class's subclasses can also use that property. That should make sense because there's an "is-a" relationship between a subclass and its superclass. For example, the above table indicates that -fx-cursor can apply to a Node object. Because a Label object is a Node (the

Label class is a descendant of the Node class), that means -fx-cursor can also apply to a Label object.

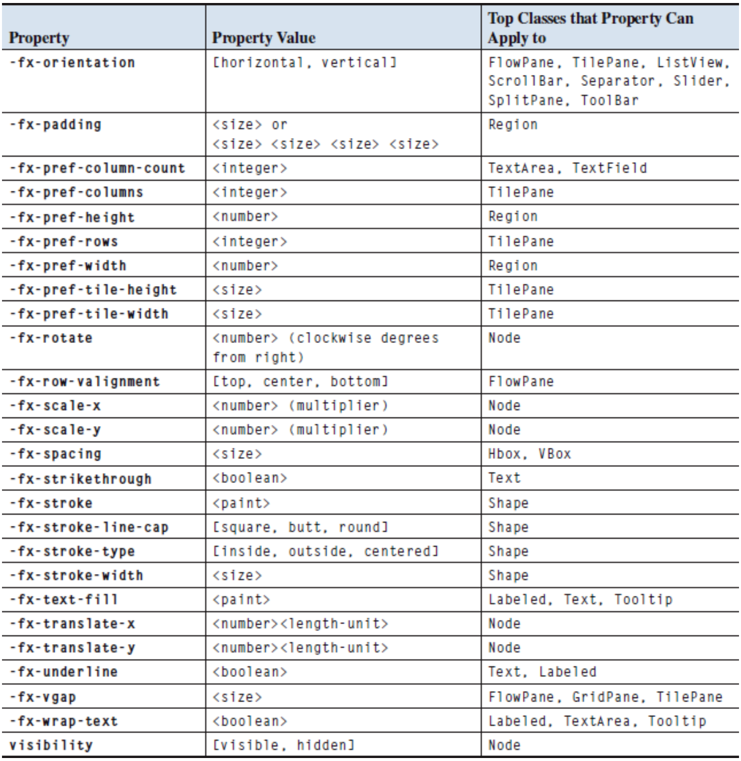
Page 816



Property	Property Value	Top Classes that Property Can Apply to
`-fx-alignment`	`[top-left, top-center, top-right, center-left, center, center-right, bottom-left, bottom-center, bottom-right]`	`Labeled, FlowPane, GridPane, Hbox, StackPane, TextField, TilePane, VBox`
`-fx-background-color`	`<paint>`	`Region`
`-fx-background-insets`	`<size>` or `<size> <size> <size> <size>`	`Region`
`-fx-border-color`	`<paint>`	`Region`
`-fx-border-insets`	`<size>` or `<size> <size> <size> <size>`	`Region`
`-fx-border-radius`	`<size>`	`Region`
`-fx-border-style`	`[none, solid, dotted, dashed]`	`Region`
`-fx-border-width`	`<size>` or `<size> <size> <size> <size>`	`Region`
`-fx-column-halignment`	`[left, center, right]`	`FlowPane`
`-fx-cursor`	`[null, crosshair, default, hand, move, e-resize, h-resize, ne-resize, nw-resize, n-resize, se-resize, sw-resize, s-resize, w-resize, v-resize, text, wait]` or `<url>`	`Node`
`-fx-effect`	`<effect>`	`Node`
`-fx-fill`	`<paint>`	`Shape`
`-fx-font`	`<font>`	`Labeled, Text, TextInputControl, Tooltip`
`-fx-font-family`	`[serif, sans-serif, cursive, fantasy, monospace]`	`Labeled, Text, TextInputControl, Tooltip`
`-fx-font-size`	`<size>`	`Labeled, Text, TextInputControl, Tooltip`
`-fx-font-style`	`[normal, italic, oblique]`	`Labeled, Text, TextInputControl, Tooltip`
`-fx-font-weight`	`[normal, bold, bolder, lighter]`	`Labeled, Text, TextInputControl, Tooltip`
`-fx-graphic`	`<url>`	`Labeled, DialogPane, Tooltip`
`-fx-graphic-text-gap`	`<size>`	`Labeled, Tooltip`
`-fx-grid-lines-visible`	`<boolean>`	`GridPane`
`-fx-hgap`	`<size>`	`FlowPane, GridPane, TilePane`
`-fx-max-height`	`<number>`	`Region`
`-fx-max-width`	`<number>`	`Region`
`-fx-min-height`	`<number>`	`Region`
`-fx-min-width`	`<number>`	`Region`

Figure 17.9a JavaFX CSS properties—part A

Page 817



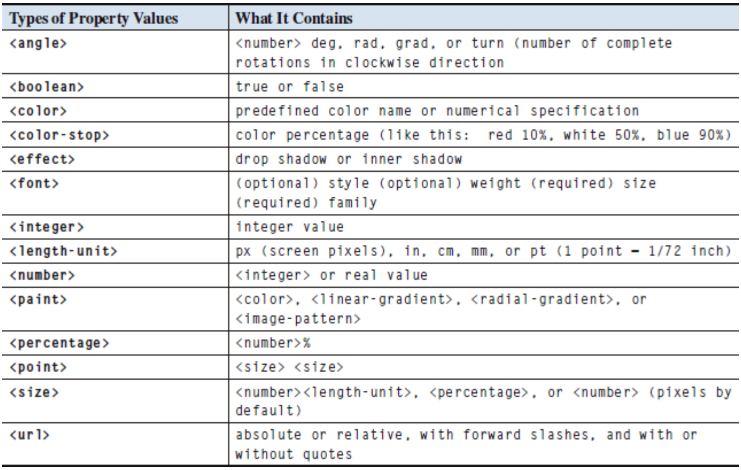
Property	Property Value	Top Classes that Property Can Apply to
`-fx-orientation`	`[horizontal, vertical]`	`FlowPane, TilePane, ListView, ScrollBar, Separator, Slider, SplitPane, ToolBar`
`-fx-padding`	`<size>` or `<size> <size> <size> <size>`	`Region`
`-fx-pref-column-count`	`<integer>`	`TextArea, TextField`
`-fx-pref-columns`	`<integer>`	`TilePane`
`-fx-pref-height`	`<number>`	`Region`
`-fx-pref-rows`	`<integer>`	`TilePane`
`-fx-pref-width`	`<number>`	`Region`
`-fx-pref-tile-height`	`<size>`	`TilePane`
`-fx-pref-tile-width`	`<size>`	`TilePane`
`-fx-rotate`	`<number>` (clockwise degrees from right)	`Node`
`-fx-row-valignment`	`[top, center, bottom]`	`FlowPane`
`-fx-scale-x`	`<number>` (multiplier)	`Node`
`-fx-scale-y`	`<number>` (multiplier)	`Node`
`-fx-spacing`	`<size>`	`Hbox, VBox`
`-fx-strikethrough`	`<boolean>`	`Text`
`-fx-stroke`	`<paint>`	`Shape`
`-fx-stroke-line-cap`	`[square, butt, round]`	`Shape`
`-fx-stroke-type`	`[inside, outside, centered]`	`Shape`
`-fx-stroke-width`	`<size>`	`Shape`
`-fx-text-fill`	`<paint>`	`Labeled, Text, Tooltip`
`-fx-translate-x`	`<number><length-unit>`	`Node`
`-fx-translate-y`	`<number><length-unit>`	`Node`
`-fx-underline`	`<boolean>`	`Text, Labeled`
`-fx-vgap`	`<size>`	`FlowPane, GridPane, TilePane`
`-fx-wrap-text`	`<boolean>`	`Labeled, TextArea, Tooltip`
`visibility`	`[visible, hidden]`	`Node`

Figure 17.9b JavaFX CSS properties—part B

Property Value Types

In Figure 17.9's JavaFX CSS properties table, for some of the values in the property value column, we showed value types rather than the actual values you'd use in a program. For example, for the -fx-background-color property, we showed <paint>, where

<paint> is a property value type. Figure 17.10 describes the different property value types.



Types of Property Values	What It Contains
`<angle>`	`<number>` deg, rad, grad, or turn (number of complete rotations in clockwise direction
`<boolean>`	true or false
`<color>`	predefined color name or numerical specification
`<color-stop>`	color percentage (like this: red 10%, white 50%, blue 90%)
`<effect>`	drop shadow or inner shadow
`<font>`	(optional) style (optional) weight (required) size (required) family
`<integer>`	integer value
`<length-unit>`	px (screen pixels), in, cm, mm, or pt (1 point - 1/72 inch)
`<number>`	`<integer>` or real value
`<paint>`	`<color>`, `<linear-gradient>`, `<radial-gradient>`, or `<image-pattern>`
`<percentage>`	`<number>`%
`<point>`	`<size>` `<size>`
`<size>`	`<number><length-unit>`, `<percentage>`, or `<number>` (pixels by default)
`<url>`	absolute or relative, with forward slashes, and with or without quotes

Figure 17.10 Different types of values for JavaFX CSS properties

Note that for the <angle> and <percentage> value types, valid values are comprised of a number and then a suffix with no whitespace between. For example, a valid <angle> value would be 90deg, and a valid <percentage> value would be 50%. Also note that for the <size> value type, one way to form a valid value is to concatenate a number with a <length-unit> with no whitespace between. For example, a valid <size> value would be 200px. If you omit the unit, then px is the default unit.

Go back to the JavaFX CSS properties table and find the properties that allow for a sequence of four <size>'s for their values. The four size values represent the property's rectangular box. For example, the -fx-border-width property specifies the widths of the four sides of the node's border. If just one size is specified, then it applies to all four sides. If four sizes are specified, those sizes refer to the top, right, bottom, and left sides, in that order.

17.11 Scene Graph Inheritance

As explained earlier, when you have more than one component in your window (which is almost always the case), you'll organize the components by adding them to a container node. For the Greeting program, we added label and text box components to a FlowPane container node. With lots of components, it's common to group related components into subcontainer nodes. All of the nodes together form the window's scene graph. Normally, when you apply a JavaFX CSS property to a node, that property applies only to that node. But some properties are special in that they get applied not only to the specified node, but also to all the nodes below that node in the scene graph's hierarchical tree of containers and components. The JavaFX documentation refers to that concept as *inheritance*. It's similar to standard inheritance. With standard inheritance, a class's members (variables and methods) flow down through its subclasses, and with this inheritance, a node's property (or properties) flows down through the nodes below it in the window's scene graph. We'll sometimes refer to it as *scene graph inheritance* to distinguish it from standard inheritance.

Some of the properties in Figure 17.9a exhibit scene graph inheritance, and here are those properties:

-fx-cursor
-fx-font
-fx-font-family
-fx-font-size
-fx-font-weight
-fx-font-style

For the above properties, the inheritance is automatic. For other properties, if you'd like a node to inherit a particular property value from its closest ancestor in the scene graph hierarchical tree, then you provide a property-value pair for that node with a value of inherit. For example, the following code causes a message label to inherit its text color from its closest ancestor:

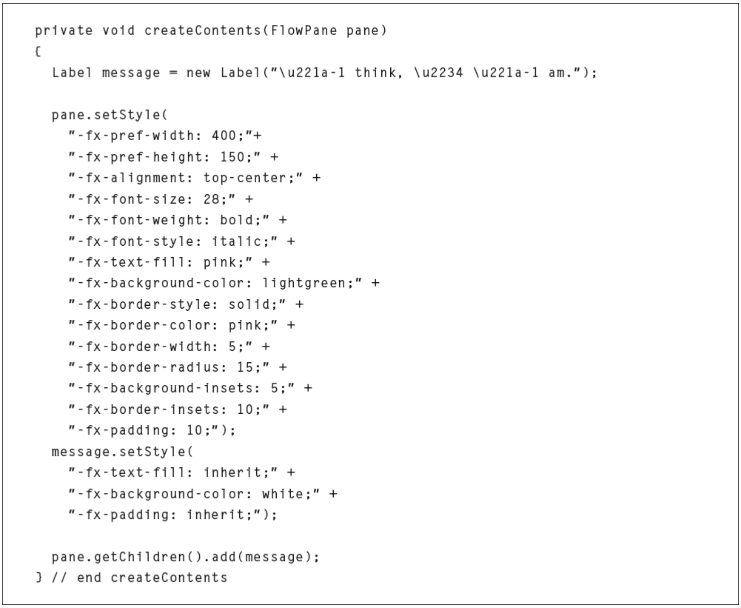
```
message.setStyle("-fx-text-fill: inherit;");
```

You might recall that in the Descartes program, we expanded the size of the label’s surrounding box by assigning values to the label’s -fx-pref-width and -fx-pref-height properties. And within that expanded box, we centered the label’s text by assigning a center value to the label’s -fx-alignment property. This time, we’re going to do things in a more scalable fashion. Our next iteration Descartes program will still have only one component (the label with Descartes’s quote), but we’re going to put most of the JavaFX CSS properties in the window’s scene graph root, so that if new nodes are added to the scene graph later on, they can rely on the styling that's already in the scene graph root.

Figure 17.11a shows the createContents method for the Descartes2 program. The rest of the program is the same as before. Note that the -fx-pref-width and -fx-pref-height property-value pairs are now in pane's setStyle method call (pane is the scene graph root) rather than in message's setStyle method call. That means the message label's surrounding box conforms to the size of its text and there's no wiggle room within it, so applying an -fx-alignment property to the label would have no effect. The solution is to expand the scene graph's root node (with -fx-pref-width and -fx-pref-height property-value pairs) so there is wiggle room between pane's box and the label's box. Thus, when we apply an -fx-alignment property to pane, the application does affect the label's position within pane's box. See the resulting label position in Figure 17.11b. It's at the top-center because we use a top-center value for pane's -fx-alignment property.

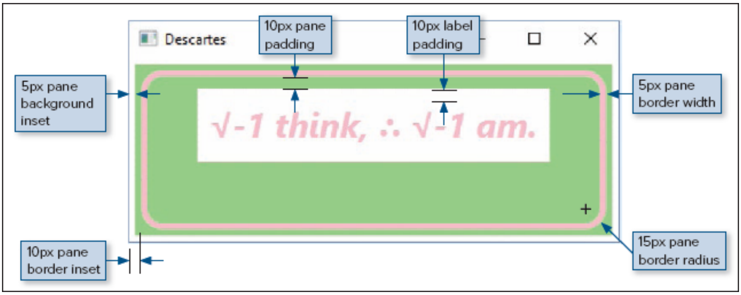
In Figure 17.11a's setStyle method call for the pane container, note how we assign font property values: 28 for the font size, bold for the weight, and italic for the style. All the font properties exhibit scene graph inheritance automatically, so as you can see in the screenshot, the message label displays its text with those characteristics.

Next up in pane's setStyle method call is the -fx-text-fill: pink property-value pair. We want the message label to inherit that pink color, but the -fx-text-fill property is not inherited by default. Thus, we have the message label call setStyle with -fx-text-fill: inherit.



```
private void createContents(FlowPane pane)
{
  Label message = new Label("\u221a-1 think, \u2234 \u221a-1 am.");

  pane.setStyle(
    "-fx-pref-width: 400;"+
    "-fx-pref-height: 150;" +
    "-fx-alignment: top-center;" +
    "-fx-font-size: 28;" +
    "-fx-font-weight: bold;" +
    "-fx-font-style: italic;" +
    "-fx-text-fill: pink;" +
    "-fx-background-color: lightgreen;" +
    "-fx-border-style: solid;" +
    "-fx-border-color: pink;" +
    "-fx-border-width: 5;" +
    "-fx-border-radius: 15;" +
    "-fx-background-insets: 5;" +
    "-fx-border-insets: 10;" +
    "-fx-padding: 10;");
  message.setStyle(
    "-fx-text-fill: inherit;" +
    "-fx-background-color: white;" +
    "-fx-padding: inherit;");

  pane.getChildren().add(message);
} // end createContents
```

Figure 17.11a Descartes2 program's createContents method


Descartes
10px pane padding
10px label padding
5px pane background inset
5px pane border width
10px pane border inset
15px pane border radius

√-1 think, ∴ √-1 am.

Figure 17.11b The window generated from the Descartes2 program

©JavaFX

Page 821

Next in pane's setStyle method call is the -fx-background-color: lightgreen property-value pair. Note the Descartes2 program's screenshot with the light green background for most of the pane, but white for the text's background color. The white comes from the message label's setStyle method call with its -fx-background-color: white property-value pair. The default value for -fx-background-color is transparent, which means that without the explicit -fx-background-color: white property-value pair, the message label would have a light green background color from the pane container's styling.

Next in pane's setStyle method call are the border property values: solid for style, pink for color, 5 for width, and 15 for radius. Note the Descartes2 program's screenshot with a solid pink 5-pixel-width border for the pane container. The border's curved corners come from the -fx-border-radius property. If there were no border radius property, the borders would have standard 90-degree corners. With a border radius property, a border's value (15 pixels in this case) defines the radius of an arc whose center is that many pixels from the corner's curved line. In Figure 17.11b, the crosshair symbol identifies the center of the border's lower-right corner arc.

Next in pane's setStyle method call are the -fx-background-insets: 5 and -fx-border-insets: 10 property-value pairs. Note the Descartes2 program's screenshot with the insets marked. In using the -fx-background-insets property, we create a gap between pane's perimeter (which matches the window's perimeter) and the area where the background color takes effect. In using the -fx-border-insets property, we create a gap between pane's perimeter and the outside of pane's border. With both insets properties, if you provide just one size value, then it applies to all four sides. If you provide four size values, those sizes refer to the top, right, bottom, and left sides, in that order. That should sound familiar because we described the same concept with the -fx-border-width property.

The last property-value pair in pane's setStyle method call is -fx-padding: 10. The padding property specifies the area on the interior of a node's border. The default is to have no padding. The padding property is for inside the border, whereas the insets properties are for outside the border. If that's hard to remember, think of a package you get in the mail. You put padding in the inside of a fragile package's box, so the padding property is for inside the border. Note the Descartes2 program's screenshot with the pane's 10-pixel padding inside of its border, and the label's 10-pixel padding inside of its (invisible) border.

The label inherits the padding property due to its setStyle method call with -fx-padding: inherit.

17.12 Style Sheets and Cascading

In Figure 17.11a's setStyle method calls, were you bothered by all the string concatenations for the property-value pairs? As an alternative to embedding JavaFX CSS property-value pairs as a huge concatenated string in a setStyle method call, you can embed them in an external file where they get linked to your Java program at run time. The JavaFX documentation refers to such external files as *style sheets*.

Style Sheets

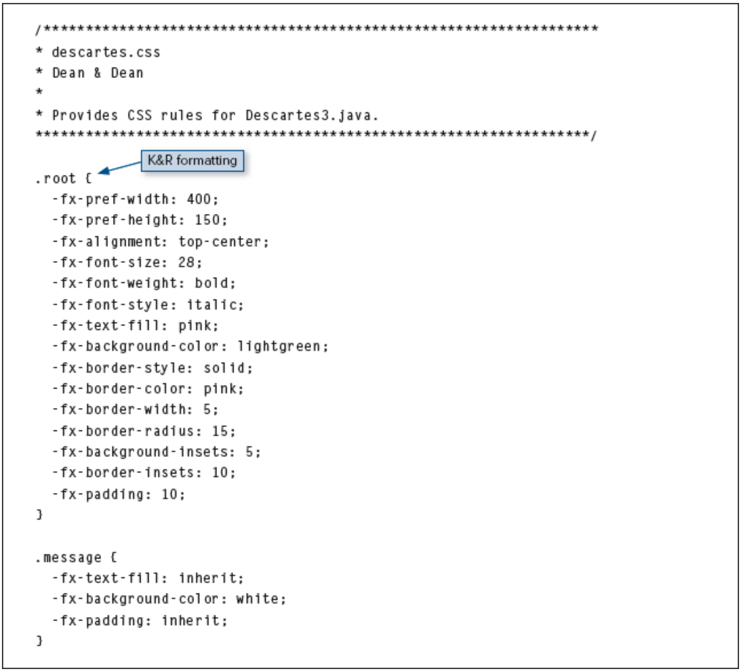
There are several advantages with using style sheets. Because a style sheet is a simple text file, not a Java file, if you change its property-value pairs, there's no need to recompile the Java program. Just run it, and it'll use the updated style sheet automatically. By moving a program's CSS property-value pairs outside of the program and into a separate file, it makes it much easier to share that formatting code with other programs. And sharing further the goals of consistency and software reuse.

There are different ways to add a style sheet to your program. You can add it to individual components or to the pane container that is the scene graph's root node, but we'll skip those approaches and stick to the more common approach of adding it to the program's Scene object. For the Descartes2 program, you might recall that the scene graph's Scene object was named scene. To link a style sheet named descartes.css to it, you would call the getStyleSheets and add methods, like this:

```
scene.getStylesheets().add("descartes.css");
```

The getStylesheets method retrieves all the style sheets already associated with the scene and adds the specified style sheet to the list. In this case, because there’s only one style sheet, the getStylesheets method retrieves an empty list and the new style sheet gets added to it.

See Figure 17.12. It shows a style sheet for a third version of the Descartes program. The prologue at the top should look familiar; CSS, like Java, uses /* and */ for comments. The property-value pairs should look familiar as well because they are the same as in the Descartes2 program.



K&R formatting

```
/*********************************************************************
* descartes.css
* Dean & Dean
*
* Provides CSS rules for Descartes3.java.
*********************************************************************/

.root {
  -fx-pref-width: 400;
  -fx-pref-height: 150;
  -fx-alignment: top-center;
  -fx-font-size: 28;
  -fx-font-weight: bold;
  -fx-font-style: italic;
  -fx-text-fill: pink;
  -fx-background-color: lightgreen;
  -fx-border-style: solid;
  -fx-border-color: pink;
  -fx-border-width: 5;
  -fx-border-radius: 15;
  -fx-background-insets: 5;
  -fx-border-insets: 10;
  -fx-padding: 10;
}

.message {
  -fx-text-fill: inherit;
  -fx-background-color: white;
  -fx-padding: inherit;
}
```

Figure 17.12 descartes.css style sheet

The file contains two rules. Each rule has a heading and a {}-surrounded block of property-value pairs. Each heading consists of a dot and then a *selector* (e.g., .root and .message). The selector's name (root and message in our example) is how the rule gets connected to a node in the JavaFX program's scene graph. For each of the two rules, note where the opening brace ({) goes—on the same line as the selector with a space between the selector and the brace. JavaFX borrows that coding convention from CSS rules in the web programming world. That coding convention is known as *K&R formatting*, where K is for Brian Kernighan and R is for Dennis Ritchie, the demigods who invented the operating system UNIX, the programming language C, and the beginning of time.5 They used the opening-brace-goes-on-the-first-line coding convention in their seminal book on C.

The first rule uses root for its selector name. The word "root" is reserved, and it tells the JVM to apply the .root rule to the program's scene graph root. If you look at the Descartes3 program in Figure 17.13, you can see that the scene graph root is pane (same as for the prior Descartes programs). Thus, the .root selector's rule (with its property-value pairs) gets applied to the pane container.

The second rule uses message for its selector name. To connect a rule that uses a nonroot selector (like .message) to a program node, the program node needs to call the getStyleClass method and then the add method. For example, in the Descartes3 program, note how we use the following line to connect the program's message label to the external file's .message rule:

```
message.getStyleClass().add("message");
```

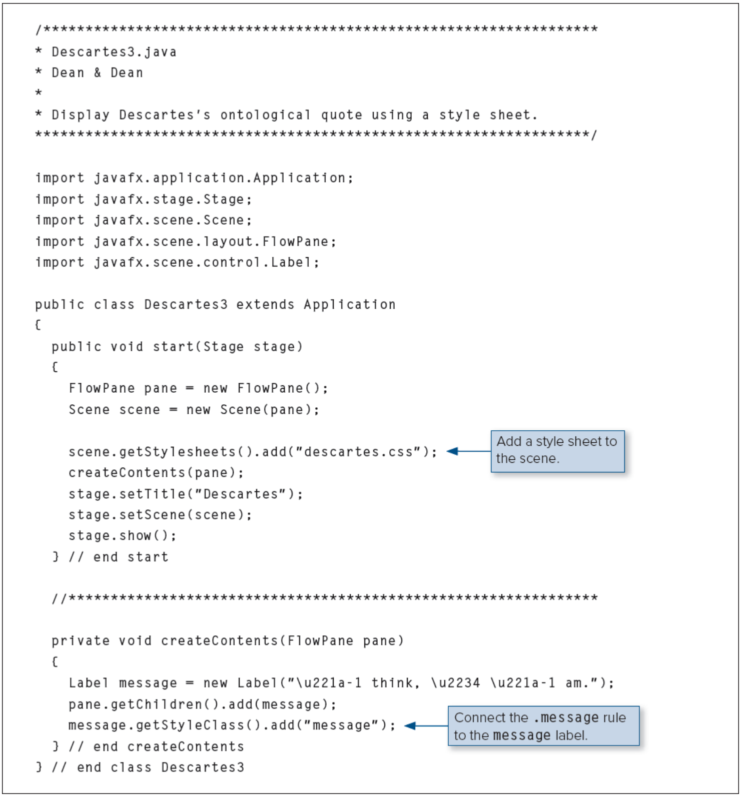
There's no requirement to use message for the selector name; we could have used any word, as long as the add method call's argument matches the selector name. But message is a good name here because its purpose is to connect a rule to the message label.

Cascading

One of the key features of CSS is the

cascading nature of how its property values are applied. As you've learned, there are different techniques for styling a JavaFX window. You can call a separate method for each property that needs to be adjusted (e.g., label.setFont(new Font(16));). Or you can use inline styles by calling setStyle. Or you can create a style sheet external file and link it to a JavaFX program by calling getStylesheets. So what happens if there's an attempt by more than one of the techniques to adjust the same property? Which technique wins when such a conflict exists? That's where cascading comes to the rescue!

If you look up the word "cascade" online or in a dictionary,6 you'll see something like "a series of stages in a process." Likewise, *Cascading* Style Sheets (CSS) uses a series of stages. There are different stages/places where values can be assigned to properties that affect a window's appearance. With multiple sets of property-value pairs organized in a staged structure, together, the whole thing is referred to as Cascading Style Sheets.



```
/******************************************************************
* Descartes3.java
* Dean & Dean
*
* Display Descartes's ontological quote using a style sheet.
******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Label;

public class Descartes3 extends Application
{
  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add("descartes.css");
    createContents(pane);
    stage.setTitle("Descartes");
    stage.setScene(scene);
    stage.show();
  } // end start

  //***************************************************************

  private void createContents(FlowPane pane)
  {
    Label message = new Label("\u221a-1 think, \u2234 \u221a-1 am.");
    pane.getChildren().add(message);
    message.getStyleClass().add("message");
  } // end createContents
} // end class Descartes3
```

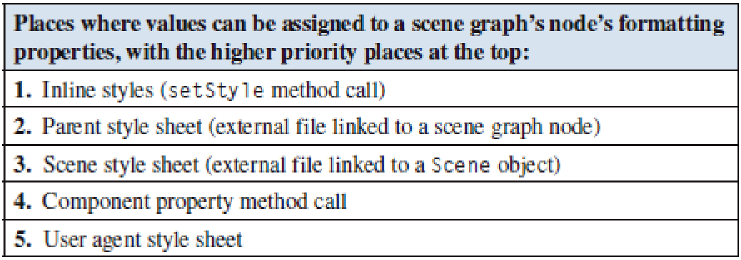
Add a style sheet to the scene.

Connect the `.message` rule to the `message` label.

Figure 17.13 Descartes3 program

Page 824

To handle the possibility of conflicting property value assignments at different places, different priorities are given to the different places. See Figure 17.14, which shows the places where property value assignments can be defined. The higher priority places are at the top, so inline styles have the highest priority.



Places where values can be assigned to a scene graph's node's formatting properties, with the higher priority places at the top:
1. Inline styles (`setStyle` method call)
2. Parent style sheet (external file linked to a scene graph node)
3. Scene style sheet (external file linked to a `Scene` object)
4. Component property method call
5. User agent style sheet

Figure 17.14 Places where property values can be assigned

After inline styles, the next highest priority place where property values can be assigned is when there's a style sheet associated with a Parent node in a scene graph. As you might recall from our earlier class hierarchy trees, the Parent class is a superclass of virtually all other classes in a scene graph, so you can think of this rung in the cascading style sheets hierarchy as whenever there's a style sheet being assigned to any node in a scene graph.

The next highest priority place where property values can be assigned is when there’s a style sheet associated with a Scene object within a scene graph. And after that is when a component calls a styling method (e.g., setFont) that adjusts a particular property for that component.

Finally, the lowest priority place where property values can be assigned is in the native style sheet settings for the platform that’s being used to run the program. As a programmer, there’s nothing you can do to modify a platform’s style sheet, so we won’t bother to describe the settings found there. But no need to worry—the settings produce run-of-the-mill standard formatting that most people are comfortable with as the default.

In displaying a node in a scene graph, the JVM will check for property value assignments that match the node, starting the search at the top of the cascading property assignment list in Figure 17.14 and continuing the search down the list, as necessary. When there is a property match, the property's value will be applied to the node, and the search down the list stops for that property.

Style Sheet Benefits, Limitations, and Workarounds

A nice feature of style sheets is that they are outside the Java code they serve. This allows you to alter the styling of an already-compiled Java program without having to modify the Java source code and without having to recompile that code. Just replace the rules in the CSS file. This is particularly helpful for mobile devices that cannot accommodate an on-board JDK and perhaps not even a JRE but are able to incorporate java programs that have been compiled into their processor’s native language. For such devices, a developer might use Java to create generic programs that have very little internal styling. That is, they minimize their use of classes like Font, Pos, and Insets, and minimize their use of method calls like setAlignment, setFont, and setStyle. To the greatest extent possible they style with CSS in external style sheets.

By using style sheets, you can give your program alternate styling options by providing alternate CSS files with different filenames. For example, from the Descartes3 program’s start method, you might recall this line:

```
scene.getStylesheets().add("descartes.css");
```

To enable dynamically loading a style sheet, change that line to this:

```
scene.getStylesheets().add(getParameters()
```

Then, when you execute your compiled java program, simply append the name of the desired .css file like this, where

# distinguishes between the different style sheet files:

```
> java Descartes3 descartes#.css
```

A program does not need to use all the style rules in a style sheet, and a program does not need to add rules before the window displays. The program can add a style rule to any scene graph node at any time during program execution. And it can add more than one rule to the same node. Each added rule becomes another element in a list of style rules possessed by that node. Given knowledge of the sequence in which style rules have been added to an object, the program can replace any previously added style rule with a different style rule. It’s like replacing a particular element in an ArrayList. Suppose you want to replace a sky node’s first (index = 0) style rule with a style rule whose selector is

.nighttime. You can do so with this code:

```
    sky.getStyleClass().set(0,
"nighttime");
```

Thus, with detailed styling specifications distributed among different style rules in one or more external style sheets, a program can adjust styling at any time during execution.

17.13 Button Control and FactorialButton Program

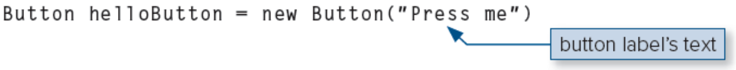
It’s now time to learn about another type of GUI control—a button.

User Interface

If you press a button on an electronic device, something usually happens. For example, if you press the power button on a television, the television turns on or off. Likewise, if you press/click a GUI *button*, something usually happens. For example, in Figure 17.1's TrustyCredit window, if you click the OK button, the entered credit card numbers go to the Trusty Credit company.

Implementation

To create a button control, call the Button constructor like this:



```
Button helloButton = new Button("Press me")
```

When this button displays, the words "Press me" appear in the center of the button. The constructor's string argument is optional. If that argument is omitted, the button displays with no writing on it.

After you have created the helloButton button, you can add it to a pane container, like this:

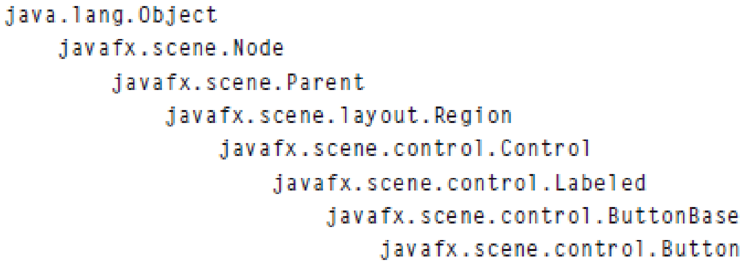
```
pane.getChildren().add(helloButton);
```

To add multiple buttons with one method, you can call the addAll method instead of the add method. To make a button useful, you'll need to implement an event handler for it. You can do that with a lambda expression or with an ordinary method registered with a method reference.

The Button class needs the javafx.scene.control package, but that may be available already, since Label and TextArea need it also. Method-reference event handling requires an additional import for ActionEvent.

Methods

If you look up the Button class on JavaFX's API website, you'll see this class hierarchy tree:



```
java.lang.Object
    javafx.scene.Node
        javafx.scene.Parent
            javafx.scene.layout.Region
                javafx.scene.control.Control
                    javafx.scene.control.Labeled
                        javafx.scene.control.ButtonBase
                            javafx.scene.control.Button
```

That should look familiar because it's identical to the Label class hierarchy tree except for the ButtonBase and Button classes at the bottom. With the same ancestor classes, Node through Labeled, the Button class inherits methods that we've already described—setFont, getText, setText, setStyle, and setVisible.

From the Node class, the Button class inherits the setEffect method, which can be used to mimic a raised or pressed button:

public final void setEffect(Effect value)

Typically, the specified argument value is an instance of either DropShadow (for a raised effect) or InnerShadow (for a lowered effect).

The Effect class is in the javafx.scene.effect package, so you'll need to import that package.

From the Region class, the Button class inherits the setPrefSize method:

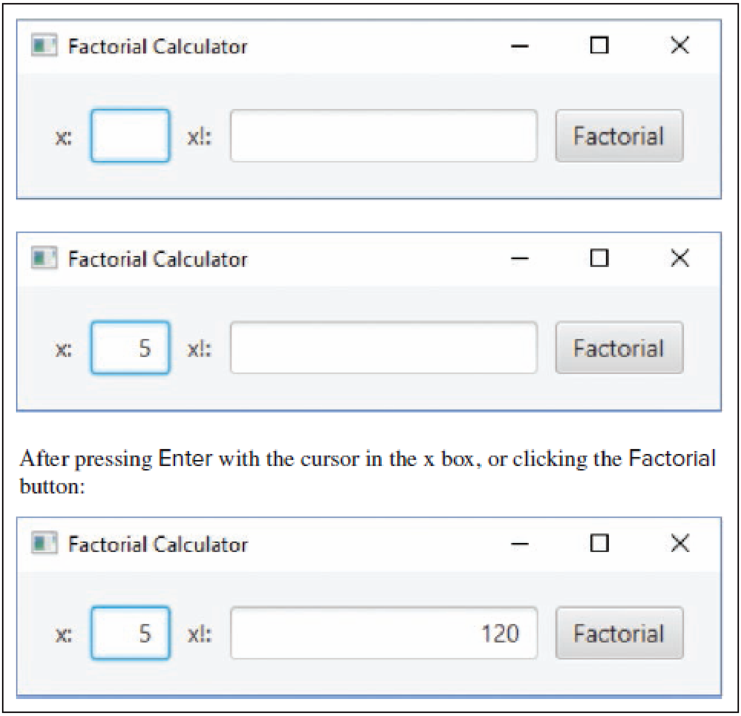
public void setPrefSize(double prefWidth, double prefHeight)

Overrides default size values.

In addition to the methods inherited from its ancestor classes Node through Labeled, the Button class inherits the setOnAction method from the ButtonBase class. You should already be comfortable with setOnAction because it's part of the TextField class, and we used it in the Greeting program. We called setOnAction with both a method reference and also a lambda expression. You've got the same options when you use it with a Button control.

FactorialButton Program

It's time to put all these button concepts into practice by showing you how it's used within a complete program. We've written a FactorialButton program that uses a Button control to calculate the factorial for a user-entered number.[7] To give you a better idea of how the program operates, see the sample session in Figure 17.15.


Factorial Calculator
x:
x!:
Factorial
Factorial Calculator
Factorial

x: 5

x!:

After pressing Enter with the cursor in the x box, or clicking the Factorial button:

Factorial Calculator
Factorial

x: 5

x!: 120

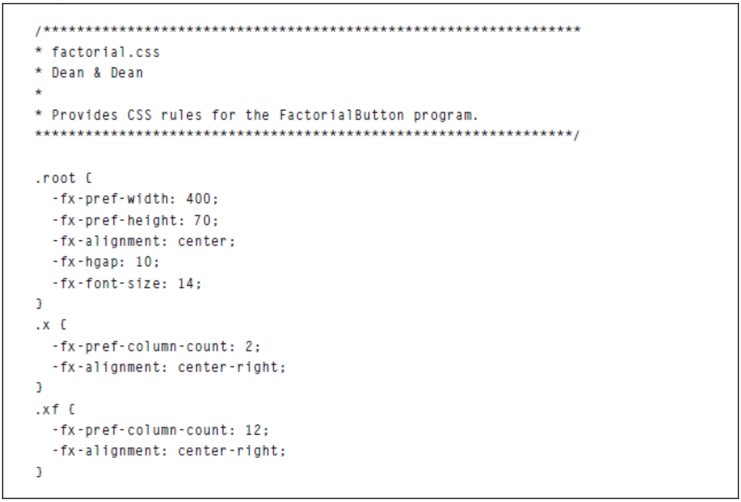
Figure 17.15 Sample session for the FactorialButton program

©JavaFX

In examining the FactorialButton program, let’s start with its style sheet. Figure 17.16 shows the style sheet’s three rules—one for the scene graph root, one for the x text box, and one for the x! text box. The .root rule’s -fx-pref-width and -fx-pref-height properties determine the window’s preferred overall width and height. “Preferred” means those are the initial dimensions (before the user possibly resizes the window) if the components fit in a window of that size. If the window is too small for the components, the window will expand accordingly. The .root rule’s -fx-alignment: center property-value pair causes the window’s components to be centered both horizontally and vertically within the window. The .root rule's -fx-hgap property determines the horizontal gap between components in the window's FlowPane container. And finally, the .root rule's -fx-font-size property determines the font size for the window's components (due to scene graph inheritance).

The .x rule uses the -fx-pref-column-count: 2 property-value pair to create a text box that holds only two characters for the input number. Why only two characters? Because the JVM would be incapable of calculating a factorial for a 3-digit number, as the result would be too large. The .xf rule uses the -fx-pref-column-count: 12 property-value pair to create a text box that holds up to twelve characters for the output number. Both the .x rule and the .xf rule use the -fx-alignment: center-right property-value pair to right align the numbers in the text boxes. Usually, strings are aligned at the left and numbers are aligned at the right, and that's what you should normally do as a developer.

Page 829



```
/*********************************************************************
* factorial.css
* Dean & Dean
*
* Provides CSS rules for the FactorialButton program.
*********************************************************************/

.root {
  -fx-pref-width: 400;
  -fx-pref-height: 70;
  -fx-alignment: center;
  -fx-hgap: 10;
  -fx-font-size: 14;
}
.x {
  -fx-pref-column-count: 2;
  -fx-alignment: center-right;
}
.xf {
  -fx-pref-column-count: 12;
  -fx-alignment: center-right;
}
```

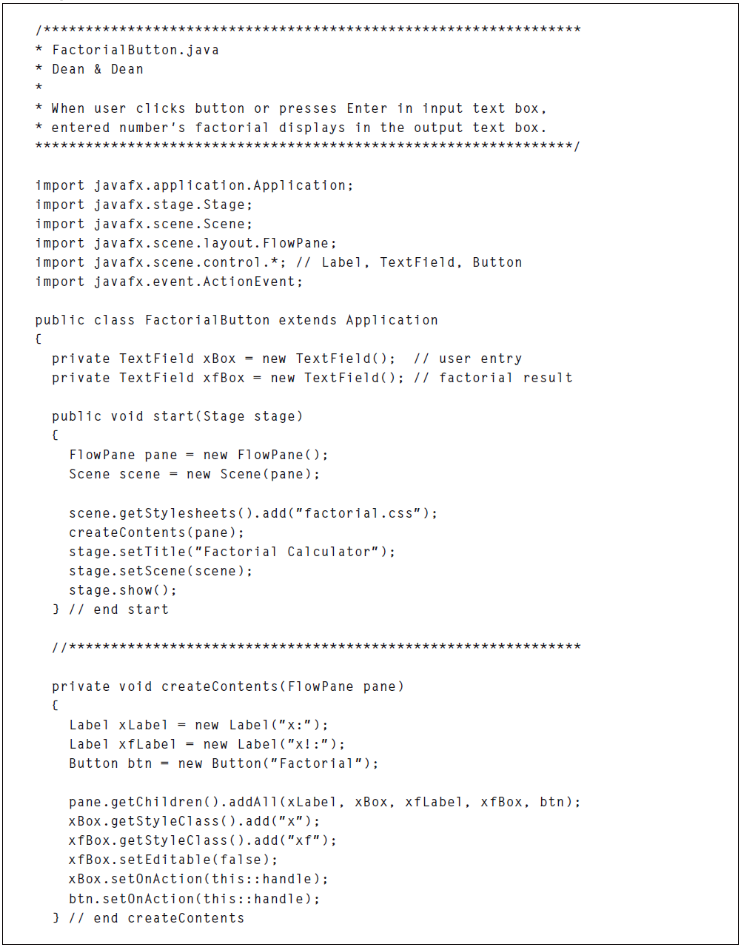
Figure 17.16 factorial.css style sheet

Figures 17.17a and 17.17b contain the FactorialButton program listing. Most of the code should already make sense because the program's structure parallels the structure in our previous GUI programs. We'll skip the more familiar code and focus on the more difficult code.

At the top of the FactorialButton class, we declare the two text boxes as instance variables because they're both used in more than one method—the createContents method and the handle method.

In the createContents method, we declare two Label controls and a Button control. Also in the createContents method, we call xfBox.setEditable(false), which prevents a user from entering anything directly into the factorial text box. That should make sense because it's up to the program (not the user) to generate the factorial. Note in the sample session how the factorial text box has a dim border. You get that visual cue free of charge whenever you call setEditable(false) from a text box control. Cool!

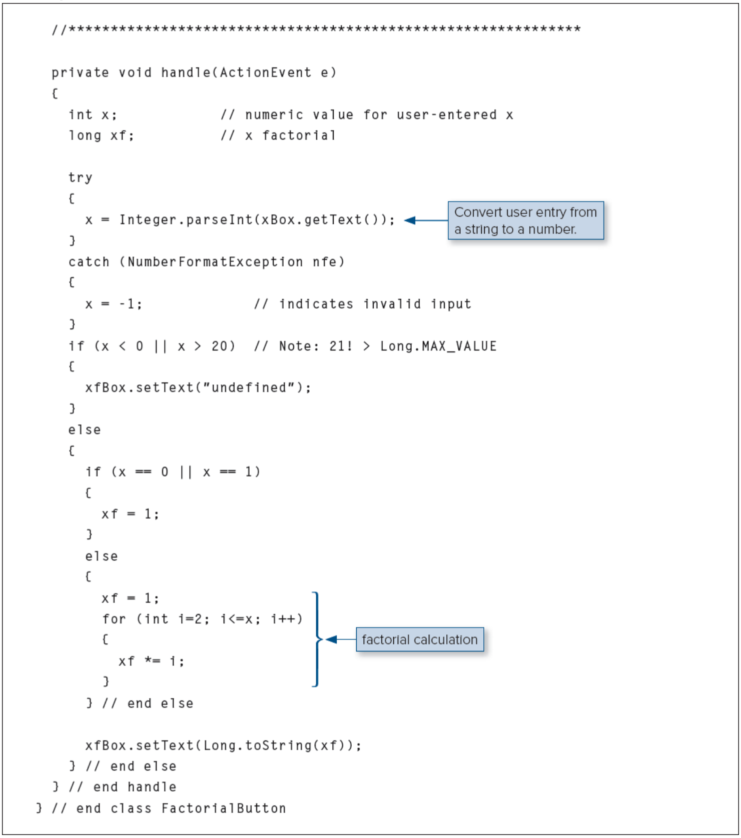
The last two statements in the createContents method use method references to register the same event-handling method with the input text box and also with the button. This gives the user two ways to trigger a response. The user can press Enter when the cursor is in the input text box or the user can click on the button. In this case, method references are the way to go. It wouldn’t make sense to use lambda expressions because then you’d have redundant code.



```
/*******************************************************************
* FactorialButton.java
* Dean & Dean
*
* When user clicks button or presses Enter in input text box,
* entered number's factorial displays in the output text box.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*; // Label, TextField, Button
import javafx.event.ActionEvent;

public class FactorialButton extends Application
{
  private TextField xBox = new TextField();  // user entry
  private TextField xfBox = new TextField(); // factorial result

  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add(”factorial.css”);
    createContents(pane);
    stage.setTitle(”Factorial Calculator”);
    stage.setScene(scene);
    stage.show();
  } // end start

  //***************************************************************

  private void createContents(FlowPane pane)
  {
    Label xLabel = new Label(”x:”);
    Label xfLabel = new Label(”x!:”);
    Button btn = new Button(”Factorial”);

    pane.getChildren().addAll(xLabel, xBox, xfLabel, xfBox, btn);
    xBox.getStyleClass().add(”x”);
    xfBox.getStyleClass().add(”xf”);
    xfBox.setEditable(false);
    xBox.setOnAction(this::handle);
    btn.setOnAction(this::handle);
  } // end createContents
```

Figure 17.17a FactorialButton program—part A

Page 831



```
//*****************************************************************

private void handle(ActionEvent e)
{
  int x;            // numeric value for user-entered x
  long xf;          // x factorial

  try
  {
    x = Integer.parseInt(xBox.getText());
  }
  catch (NumberFormatException nfe)
  {
    x = -1;             // indicates invalid input
  }
  if (x < 0 || x > 20)  // Note: 21! > Long.MAX_VALUE
  {
    xfBox.setText("undefined");
  }
  else
  {
    if (x == 0 || x == 1)
    {
      xf = 1;
    }
    else
    {
      xf = 1;
      for (int i=2; i<=x; i++)
      {
        xf *= i;
      }
    } // end else

    xfBox.setText(Long.toString(xf));
  } // end else
} // end handle
} // end class FactorialButton
```

Convert user entry from a string to a number.

factorial calculation

Figure 17.17b FactorialButton program—part B

Page 832

Figure 17.17b's handle method is chock full of interesting code. First, note that xf, the local variable that stores x's factorial, is declared to be a long, not an int. That's because a factorial can become very large very quickly.

In the try block, note the Integer.parseInt method call. If you ever need to read numbers or display numbers in a GUI program, you must use string versions of the numbers. Thus, to read a number from the input text box, we first read it in as a string, and then we convert the string to a number. To accomplish this, we read the string using xBox.getText(), and then we convert it to a number using Integer.parseInt.

Ideally, you should always check user input to make sure it's valid. In our handle method, we check for three types of invalid input—a noninteger input, a negative number input, and an excessively large input. The Integer.parseInt method call checks that the input characters form an integer. If they don’t, the catch block’s assignment gives x a value that will make it fail the next test. The next test displays “undefined” in xfBox’s viewing area if the input number is either negative or larger than the largest input (20) whose factorial can fit into a Long. (Long.MAX_VALUE = 9223372036854775807 is indeed a very large number!) If the result is not “undefined,” the handle method proceeds to calculate the factorial. Then it displays the result in xfBox’s viewing area.

After validating the input, the handle method calculates the factorial. It first takes care of the special case when x equals 0 or 1. It then takes care of the x ≥ 2 case by using a for loop. Study the code. It works fine, but do you see a way to make it more compact? You can omit the block of code that starts with if (x = = 0 || x = = 1) because that case is handled by the else block. More specifically, you can delete the six lines above the second xf = 1; line.

17.14 Distinguishing Between Multiple Events

Before now, the event handler's ActionEvent parameter didn't do anything. It was just a minor annoyance. In this section you'll see examples where the ActionEvent parameter's object is useful. In particular, you'll see how we use it to retrieve the component that fired the event and then use that component to distinguish between different possible events.

Using getSource with the Button's Reference Name

Suppose you register the same event-handling method with two components as we did in the preceding FactorialButton program. But suppose you want the response to depend on which component fired the event. To respond appropriately you must determine which component fired the event. Then you can customize your event handling: Do one thing if component X fired the event, and do another thing if component Y fired the event.

From within an event-handling method, how can you determine the source of an event? In other words, how can you identify the component that fired an event? Call getSource, of course! More specifically, within a handle method, use its ActionEvent parameter to call getSource. The getSource method returns a reference to the component whose event fired. To see which component that was, you can use == to compare the returned reference with a particular component reference. For example, for the FactorialButton program, if you'd like to force the user to click the button rather than press Enter in the input text box, you could add this code:

```
Alert alert;      // warn the user to click the button

if (e.getSource() == xBox)
{
  alert = new Alert(Alert.AlertType.WARNING);
  alert.setTitle("Alert");
  alert.setHeaderText("");
  alert.setContentText(
    "Click factorial button to perform operation.");
  alert.showAndWait();
} // end if
```

The Alert class implements a dialog box that forces the user to respond before going further. It's defined with the other controls in the javafx.scene.control package. To learn more about the Alert dialog and other dialogs, such as the ones created with TextInputDialog and ChoiceDialog, refer back to the optional GUI section at the end of Chapter 3.

Using getSource with the

Button's Label

In the previous example, to determine which component was responsible for an event being fired, the event handler compared the event's source to a particular button's reference variable. That works most of the time, but not always. Suppose there's no reference variable name that the event handler can access. That happens when the component is declared in a different class from the class that contains the event-handler method.

Another situation where comparing the event's source to a reference variable doesn't work is when you need a component to be *modal*. A modal component is a component with more than one state or status. For example, suppose there's a button whose label toggles between "Show Details" and "Hide Details." The two labels correspond to two different modes of operation—in one mode details are shown, and in another mode details are hidden. If a modal button is clicked, getSource can retrieve the button, but it cannot retrieve the button's mode. To retrieve the button's mode, the trick is to retrieve the button's label. In the following code fragment, note how we retrieve the button with e.getSource and retrieve the button's label with getText. We then compare the retrieved label with "Show Details." If they're equal, we assign a string to the instructions label that describes how to install a flapper to fix a toilet tank that continually runs. Also, we toggle the button's label to "Hide Details." If they're unequal, we assign the empty string to the instruction label and toggle the button's label to "Show Details."

```
private void handle(ActionEvent e)
{
  Button btn = (Button) e.getSource();
  if (btn.getText().equals(“Show Details”))
  {
    instructions.setText(“Buy a toilet tank flapper.” +
      ” Remove old flapper. Mount new flapper”);
    btn.setText(“Hide Details”);
  }
  else
  {
    instructions.setText(“”);
    btn.setText(“Show Details”);
  }
} // end handle
```

getEventType Method

In addition to knowing the source of an event, you may also want to know the type of that event. If your event-handling method employs an ActionEvent parameter, you can use e.getEventType().getName() to retrieve the name of the event type as a string. If your event-handling method employs some kind of InputEvent parameter, like a KeyEvent or a MouseEvent, you can use e.getEventType().toString() to retrieve a string value like

KEY_PRESSED, KEY_RELEASED, MOUSE_PRESSED, MOUSE_RELEASED, MOUSE_DRAGGED, or MOUSE_CLICKED. Then you can compare the returned value in an if statement or a switch construct and take appropriate action. Later on in this chapter, we provide an example that illustrates these concepts.

17.15 Colors

As mentioned earlier, customers typically won't buy programs unless they are GUI based. Why? Because GUI programs can lead to a better user experience with improved functionality. Part of that better user experience comes from the proper use of color. For example, consistent color patterns can provide visual cues for input prompts and warning messages, which can speed up the user input process. Color can further enhance the user's experience just by being itself. Remember, color is fun!

To specify a color value, you can use one of these five different formats:

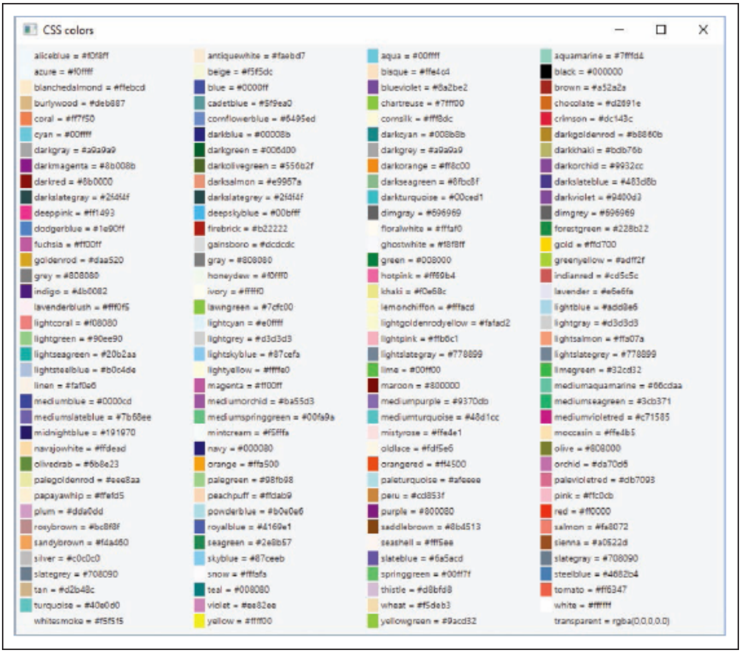
> Color name – for example, chartreuse
>
> RGB value – specifies amounts of red, green, and blue
>
> RGBA value – specifies red, green, and blue, plus amount of opacity
>
> HSB value – specifies amounts of hue, saturation, and brightness
>
> HSBA value – specifies hue, saturation, and brightness, plus amount of opacity

The first format, using a color name, is what we used in our prior examples. Specifically, we used the color names white, lightgreen, and pink to specify color values for the properties -fx-text-fill, -fx-background-color, and -fx-border-color. Figure 17.18 shows all the color names defined by the JavaFX CSS specification. You can see that color names use all lowercase letters with no spaces between multiple-word names. For example, for your applications that involve lightly boiled almonds, you'll want to use the blanchedalmond color value. JavaFX's color names match the color names in the web programming CSS specification. In the next chapter, there's an exercise that asks you to write a program that generates Figure 17.18's screenshot. We postpone the exercise until the next chapter because the program relies on layout concepts introduced in that chapter.

RGB Values

If you can't find the perfect color with one of the color names, as an alternative, you can form a color by blending specified amounts of red, green, and blue. To specify an amount of a color, you can use a percentage, an integer, or a hexadecimal number (we'll explain hexadecimal shortly). Here are the allowable ranges for each technique:

> Percentage – 0% to 100% for each color
>
> Integer – 0 to 255 for each color
>
> Hexadecimal – 00 to ff for each color



aliceblue = #f0f8ff
antiquewhite = #faebd7
aqua = #00ffff
aquamarine = #7fffd4
azure = #f0ffff
beige = #f5f5dc
bisque = #ffe4c4
black = #000000
blanchedalmond = #ffebcd
blue = #0000ff
blueviolet = #8a2be2
brown = #a52a2a
burlywood = #deb887
cadetblue = #5f9ea0
chartreuse = #7fff00
chocolate = #d2691e
coral = #ff7f50
cornflowerblue = #6495ed
cornsilk = #fff8dc
crimson = #dc143c
cyan = #00ffff
darkblue = #00008b
darkcyan = #008b8b
darkgoldenrod = #b8860b
darkgray = #a9a9a9
darkgreen = #006400
darkgrey = #a9a9a9
darkkhaki = #bdb76b
darkmagenta = #8b008b
darkolivegreen = #556b2f
darkorange = #ff8c00
darkorchid = #9932cc
darkred = #8b0000
darksalmon = #e9967a
darkseagreen = #8fbc8f
darkslateblue = #483d8b
darkslategray = #2f4f4f
darkslategrey = #2f4f4f
darkturquoise = #00ced1
darkviolet = #9400d3
deeppink = #ff1493
deepskyblue = #00bfff
dimgray = #696969
dimgrey = #696969
dodgerblue = #1e90ff
firebrick = #b22222
floralwhite = #fffaf0
forestgreen = #228b22
fuchsia = #ff00ff
gainsboro = #dcdcdc
ghostwhite = #f8f8ff
gold = #ffd700
goldenrod = #daa520
gray = #808080
green = #008000
greenyellow = #adff2f
grey = #808080
honeydew = #f0fff0
hotpink = #ff69b4
indianred = #cd5c5c
indigo = #4b0082
ivory = #fffff0
khaki = #f0e68c
lavender = #e6e6fa
lavenderblush = #fff0f5
lawngreen = #7cfc00
lemonchiffon = #fffacd
lightblue = #add8e6
lightcoral = #f08080
lightcyan = #e0ffff
lightgoldenrodyellow = #fafad2
lightgray = #d3d3d3
lightgreen = #90ee90
lightgrey = #d3d3d3
lightpink = #ffb6c1
lightsalmon = #ffa07a
lightseagreen = #20b2aa
lightskyblue = #87cefa
lightslategray = #778899
lightslategrey = #778899
lightsteelblue = #b0c4de
lightyellow = #ffffe0
lime = #00ff00
limegreen = #32cd32
linen = #faf0e6
magenta = #ff00ff
maroon = #800000
mediumaquamarine = #66cdaa
mediumblue = #0000cd
mediumorchid = #ba55d3
mediumpurple = #9370db
mediumseagreen = #3cb371
mediumslateblue = #7b68ee
mediumspringgreen = #00fa9a
mediumturquoise = #48d1cc
mediumvioletred = #c71585
midnightblue = #191970
mintcream = #f5fffa
mistyrose = #ffe4e1
moccasin = #ffe4b5
navajowhite = #ffdead
navy = #000080
oldlace = #fdf5e6
olive = #808000
olivedrab = #6b8e23
orange = #ffa500
orangered = #ff4500
orchid = #da70d6
palegoldenrod = #eee8aa
palegreen = #98fb98
paleturquoise = #afeeee
palevioletred = #db7093
papayawhip = #ffefd5
peachpuff = #ffdab9
peru = #cd853f
pink = #ffc0cb
plum = #dda0dd
powderblue = #b0e0e6
purple = #800080
red = #ff0000
rosybrown = #bc8f8f
royalblue = #4169e1
saddlebrown = #8b4513
salmon = #fa8072
sandybrown = #f4a460
seagreen = #2e8b57
seashell = #fff5ee
sienna = #a0522d
silver = #c0c0c0
skyblue = #87ceeb
slateblue = #6a5acd
slategray = #708090
slategrey = #708090
snow = #fffafa
springgreen = #00ff7f
steelblue = #4682b4
tan = #d2b48c
teal = #008080
thistle = #d8bfd8
tomato = #ff6347
turquoise = #40e0d0
violet = #ee82ee
wheat = #f5deb3
white = #ffffff
whitesmoke = #f5f5f5
yellow = #ffff00
yellowgreen = #9acd32
transparent = rgba(0,0,0,0.0)

Figure 17.18 Color names

©JavaFX

Page 835

This technique of creating a color by mixing specified amounts of red, green, and blue is used by many programming languages. The red, green, blue 3-tuple is commonly referred to as an *RGB value*. To specify an RGB value with percentages, use this format:

rgb(*red-percent*,*green-percent*,*blue-*

percent)

Each percentage value must be between 0% and 100%. Here's an example style sheet rule that uses an RGB value with percentages:

```
.root {-fx-background-color: rgb(56%,93%,56%);}
```

What background color does the above rule generate? 93% for the second value means that it uses quite a bit of green in the color mixture. With all three values above 50%, that means that the resulting color is fairly light in terms of its shade. Thus, the rule produces a light green color; it's equivalent to what's produced by the lightgreen color name used in the Descartes3 program.

In the above style sheet rule, note that the entire rule is on one line. Previously we said to use K&R formatting for style sheet rules, where the opening brace goes on the same line as the selector with the property-value pairs and the closing brace on separate lines. But for short rules, coding conventions suggest that you have a choice. Free to use K&R, but for a space-saving alternative, feel free to put the entire rule on one line.

White light is the combination of all colors,[8] so rgb(100%,100%,100%) produces white. Black is the absence of all colors, so rgb(0%,0%,0%) produces black.

To specify an RGB value with integers, use this format:

rgb(*red-integer,green-integer,blue-integer*)

Each integer value must be between 0 and 255. Here's an example style sheet rule that uses an RGB value with integers:

```
.root {-fx-border-color: rgb(255,192,203);}
```

What color does the above rule generate? 255 for the first value means that it uses quite a bit of red in the color mixture. With green and blue above the halfway point at 192 and 203, respectively, the rule produces a light shade of red; it's equivalent to what's produced by the pink color name used in the Descartes3 program.

As you learned in the Type Details chapter, hexadecimal numbers are formed with the 16 symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. To specify an RGB value with hexadecimal values, you'll need to use the format #rrggbb, where:

rr = two hexadecimal digits that specify the amount of red

gg = two hexadecimal digits that specify the amount of green

bb = two hexadecimal digits that specify the amount of blue

Here's an example style sheet rule that uses a hexadecimal color value:

.root {-fx-border-color: #ffc0cb;}

Hexadecimal symbols are case-insensitive, so ff (as well as FF) indicates the maximum amount of red. The c0 and cb values supply a considerable amount of the other colors, and the result is the same pink color that's produced by the rgb(255,192,203) value in the prior example.

Opacity Values

You've learned about the rgb construct; now it's time to learn about the rgba construct. The rgba construct uses the same red, green, and blue values for its color, and adds a fourth value for opacity. Opacity refers to the inability to see through something. It’s the opposite of transparency. If the opacity value is 100%, that means the color is completely opaque, and if there is content behind the color, that content gets covered up. At the other extreme, if the opacity value is 0%, that means the color is completely transparent. Using an opacity value between the two extremes causes the component’s background color to blend with the background color of that component’s container.

To specify an RGBA value, use one of these two formats:

rgba(*red-integer,green-integer,blue-integer,opacity-number-between-0-and-1*)
rgba(*red-percent,green-percent,blue-percent,opacity-number-between-0-and-1*)

The opacity value must be in the form of a decimal number between 0 and 1, with 0 being completely transparent, 1 being completely opaque, and .5 in between. For the first format above, each integer value must be between 0 and 255, with 0 providing the least intensity and 255 providing the most. That should sound familiar because that was also the case for integers with the rgb construct. For the second format above, each percent value must be between 0% and 100%.

Here's an example style sheet rule that uses an RGBA color value:

```
.message {-fx-text-fill: rgba(127,127,255,.5);}
```

With 255 for the blue value and an opacity value of .5, the JVM displays the message's text with a 50/50 blend of the specified blue color and the container's background color.

HSB and HSBA Values

Sometimes there are multiple ways to accomplish the same thing. You've already learned how to specify color values with names, the rgb construct, and the rgba construct. Now for the hsb construct. Here's the syntax:

hsb(*hue-integer,saturation-percent,lightness-percent*)

HSB stands for hue, saturation, and brightness. The first value in the hsb construct is the color's hue, as represented by the position in degrees on the color wheel shown in Figure 17.19. As you can see, 0 degrees is for red, 120 degrees is for green, and 240 degrees is for blue. For a circle, 0 degrees is equivalent to 360 degrees. So to specify red, you can use 360 as an alternative to 0.

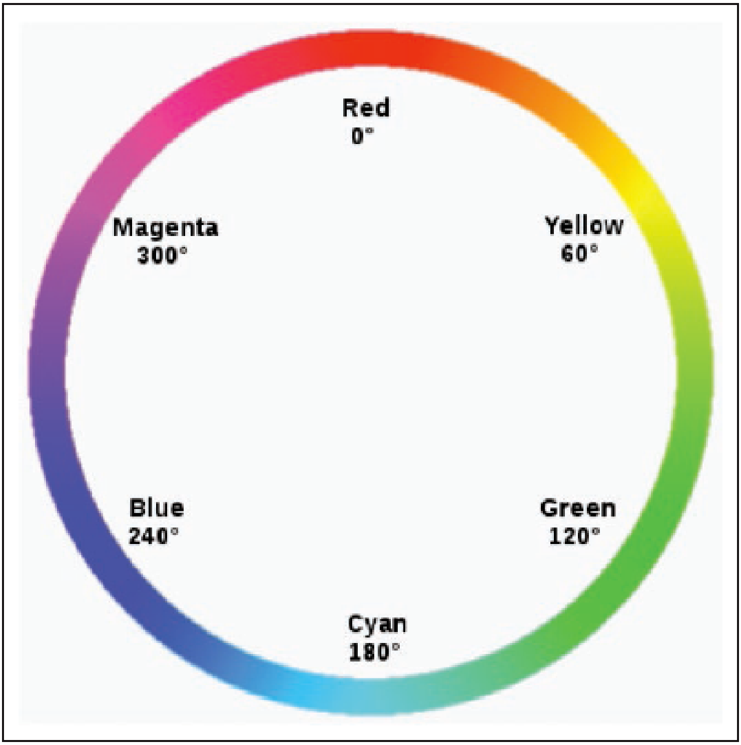

Red
0°
Yellow
60°
Magenta
300°
Blue
240°
Green
120°
Cyan
180°

Figure 17.19 Color wheel for HSB and HSBA hue values

The second value in the hsb construct is the color’s percentage of saturation. The larger the saturation value, the closer the color is to its full color. The smaller the saturation value, the closer the color is to gray.

The third value in the hsb construct is the color’s percentage of brightness.

A 0% brightness value generates black, regardless of the values for hue and saturation. The effect of a 100% brightness value is dependent on the saturation value. With a saturation value of 0%, 100% brightness produces white. With a saturation value of 100%, 100% brightness produces a very bright version of the color.

Here's an example style sheet rule that uses an HSB color value:

```
.root {-fx-background-color: hsb(120,39%,93%);}
```

The first value, 120, generates a green hue. The second and third values temper the green so the result is a light green color; it's equivalent to what's produced by the lightgreen color name used in the Descartes3 program.

Previously, you learned how to add transparency to an RGB value by using the rgba construct. Likewise, to add transparency to an HSB value, you can use the hsba construct. Here's the syntax:

hsba(*hue-integer,saturation-percent,lightness-percent,opacity-number-between-0-and-1*)

The fourth argument specifies the opacity. The opacity value must be in the form of a decimal number between 0 and 1, with 0 being completely transparent and 1 being completely opaque.

When coming up with RGB and HSL values for your programs, it's perfectly acceptable to use trial and error, but to save time, you may want to use an online color picker. For example, check out http://colorizer.org.

17.16 ColorChooser Program

Let's put some of what you've learned

about event handlers and colors into practice in the context of a complete program. In our ColorChooser program, we implement Stop and Go buttons that set the window's background color to light red (for stop) or light green (for go), respectively. See Figure 17.20 to get an idea of what we're talking about.

See the colorChooser.css file in Figure 17.21. In particular, note the color properties. To achieve a light blue background for the window, we apply -fx-background-color: rgb(245,245,255) to the scene graph root. For the two buttons' background colors, we apply -fx-background-color: red and -fx-background-color: green, respectively.

Button labels are black by default. For the ColorChooser program, we want white labels for the buttons to make them stand out from the buttons' red and green backgrounds. We could implement white labels by adding -fx-text-fill: white property-value pairs to both button rules, but then we'd have redundant code. As an alternative, we could add that property-value pair to the .root rule and have the buttons inherit that feature by adding -fx-text-fill: inherit to the two button rules. That would be reasonably elegant, but there's a better way. Here's the relevant code from the colorChooser.css style sheet:

```
Button {-fx-text-fill: white;}
```

Note that the rule uses Button for its selector. Previously, all our JavaFX CSS selectors have started with a dot (e.g., .root, .goButton). Dot-prefixed selectors are formally referred to as *class selectors*. You should recall that those selectors are tied to a component in the pane's scene graph by making this call: *component*.getStyleClass().add("*selector*").

On the other hand, selectors without the dot (e.g., Button) are referred to as *type selectors*. Type selectors match a particular “type” of node in the pane’s scene graph. The node’s type is defined by its class name. So the Button selector matches all Button components that are in the pane’s scene graph. Thus, the rule causes both buttons to use white for their labels.

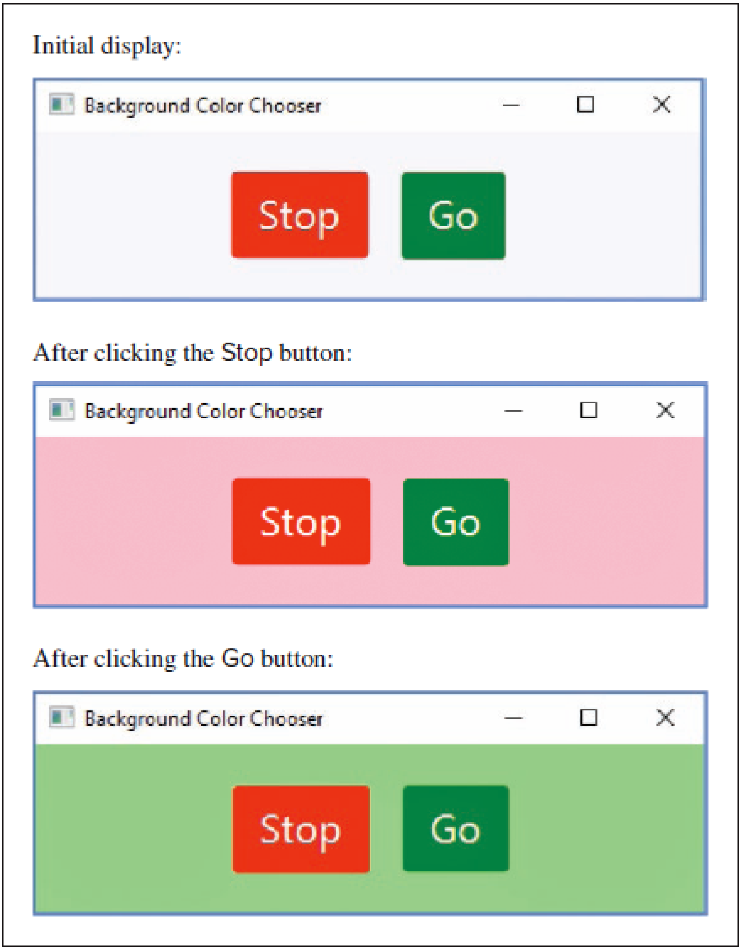
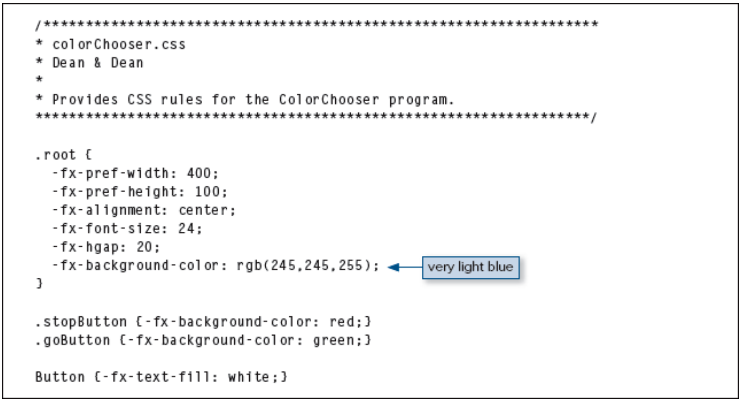

Initial display:
Background Color Chooser
Stop
Go
After clicking the Stop button:
Background Color Chooser
Stop
Go
After clicking the Go button:
Background Color Chooser
Stop
Go

Figure 17.20 Sample session for the ColorChooser program

©JavaFX

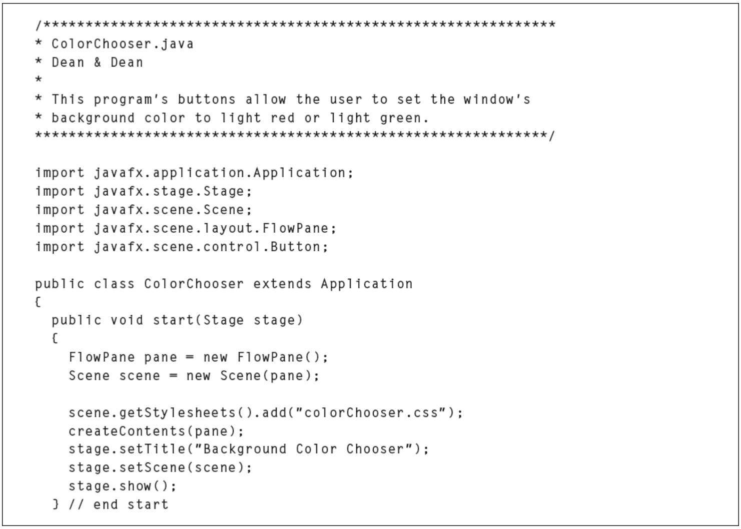
See the ColorChooser program in Figures 17.22a and 17.22b. Most of the code should already make sense because its structure mirrors the structure in our previous GUI programs. We'll focus on the new code—the event-handler code. With different operations needed for the two buttons, there's no avoidance-of-redundancy benefit in using method references with a separate event-handler method. So instead we use the more compact lambda expression technique.



```
/*********************************************************************
* colorChooser.css
* Dean & Dean
*
* Provides CSS rules for the ColorChooser program.
*********************************************************************/

.root {
  -fx-pref-width: 400;
  -fx-pref-height: 100;
  -fx-alignment: center;
  -fx-font-size: 24;
  -fx-hgap: 20;
  -fx-background-color: rgb(245,245,255);
}

.stopButton {-fx-background-color: red;}
.goButton {-fx-background-color: green;}

Button {-fx-text-fill: white;}
```

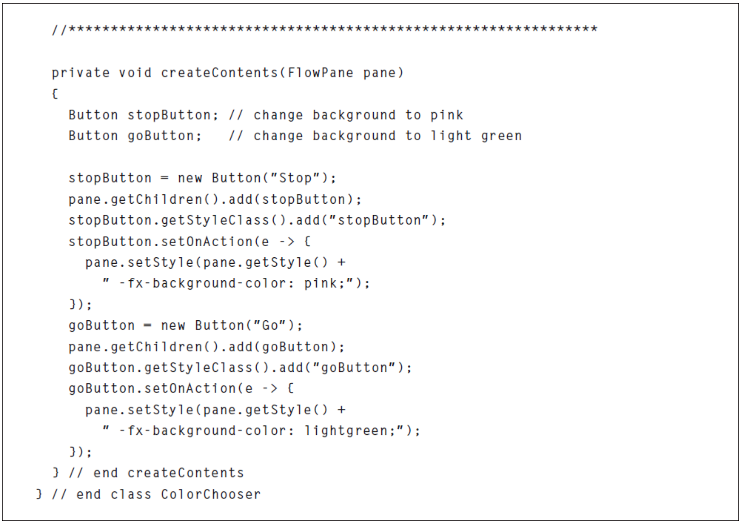
very light blue

Figure 17.21 colorChooser.css style sheet



```
/*************************************************************
* ColorChooser.java
* Dean & Dean
*
* This program’s buttons allow the user to set the window’s
* background color to light red or light green.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Button;

public class ColorChooser extends Application
{
  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add(”colorChooser.css”);
    createContents(pane);
    stage.setTitle(”Background Color Chooser”);
    stage.setScene(scene);
    stage.show();
  } // end start
```

Figure 17.22a ColorChooser program—part A



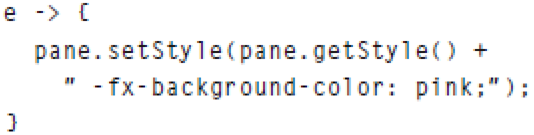
```
  //*************************************************************

  private void createContents(FlowPane pane)
  {
    Button stopButton; // change background to pink
    Button goButton;   // change background to light green

    stopButton = new Button(”Stop”);
    pane.getChildren().add(stopButton);
    stopButton.getStyleClass().add(”stopButton”);
    stopButton.setOnAction(e -> {
      pane.setStyle(pane.getStyle() +
        ” -fx-background-color: pink;”);
    });
    goButton = new Button(”Go”);
    pane.getChildren().add(goButton);
    goButton.getStyleClass().add(”goButton”);
    goButton.setOnAction(e -> {
      pane.setStyle(pane.getStyle() +
        ” -fx-background-color: lightgreen;”);
    });
  } // end createContents
} // end class ColorChooser
```

Figure 17.22b ColorChooser program—part B

Page 841

In the createContents method, the

stop button calls the setOnAction method with this lambda expression for its argument:



```
e -> {
  pane.setStyle(pane.getStyle() +
    " -fx-background-color: pink;");
}
```

Note how we use setStyle to apply a new style rule to the pane container. If you refer back to the colorChooser.css file, you can see the pane container’s original style rule, which uses the .root selector. That original style rule assigns light blue (with rgb(245,245,255)) to the background color. In the above lambda expression, the pane container’s new style rule is a concatenation of the pane container’s current style rule plus a property-value pair that assigns pink to the background color. Uh oh. That means the pane container will have conflicting background color assignments—light blue first, and then pink. Not to worry. When there’s such a conflict, the later assignment wins.

Thus, the background color gets changed to pink.

Study the go button's setOnAction method call on your own. You can see that its lambda expression parallels the stop button's lambda expression.

Summary

- The Application class should be used as the superclass for all of your JavaFX GUI application windows.
- When Application subclass programs load, they call their start methods automatically, so the start method can be used as a starting point, and the main method is unnecessary.
- The Stage class implements all the standard window features such as a border, a title bar, a minimize

button, a close-window button (the “X”), the ability to resize the window, and so on.

- Label is a read-only control; the user simply reads the label’s message.
- The TextField control allows the user to enter text into a text box.
- When the user interacts with a component (e.g., when the user clicks a button or presses enter while in a text box), the component fires an event.
- If a component has an event handler attached to it, the fired event is “heard” by the event handler and its instructions are executed.
- With JavaFX, you can implement event handlers with method references or lambda expressions.
- Property binding allows you to

update a target component simultaneously as a source component changes.

- JavaFX CSS properties come from and very closely match the properties found in web programming's CSS standard.

- Scene graph inheritance is when a JavaFX CSS property gets applied not only to a specified node, but also to all the nodes below that node in the scene graph's hierarchical tree of containers and components.

- Cascading is the concept of JavaFX CSS properties being defined at different places and how those assignments are given precedence.

- There are different techniques for distinguishing between multiple events—using getSource with a button's reference name or label, or using getEventType to determine the event's type.

- To specify a color, you can use a name, an RGB value, an RGBA value, an HSB value, or an HSBA value.

Review Questions

§17.1 Introduction

1. Why are Swing components considered to be lightweight (as opposed to AWT components, which are considered to be heavyweight)?

2. What is an RIA?

3. Provide several reasons why JavaFX is considered to be an improvement over Swing and AWT?

§17.2 SimpleWindow Program

4. What class should be used as a superclass for your JavaFX programs?

5. How is the start method special?

§17.3 Scene and Stage

6. What are two ways to cause a window to be a certain size?

7. To position a program's window on the computer screen, you can call the Stage object's setX and setY methods with x and y argument values that specify the x, y coordinate position of the window's top-left corner. (T / F)

§17.5 Label Control

Page 843

8. In a single statement, declare a Label reference variable named hello and initialize it with the string "Hello World!".

9. If you'd like to limit a Label control to a width narrower than its surrounding container, what two methods should you call?

§17.6 TextField Control

10. Provide a statement that reduces a text box's current preferred column count by two. Assume the text box's name is nameBox. Hint: As you might expect, there is also a getPrefColumnCount method.

11. How is the TextField class's setPromptText method different from its setText method?

§17.7 Greeting Program

12. If you have only one component, you can add it directly to a Scene object, but not with multiple components. How can you add multiple components to a scene?

§17.8 Event Handling

13. What is an event?

14. Provide a statement that uses setOnAction to register an event handler named verify with a Button control named verifyButton.

15. If you want to reuse the code for an event handler, which is better—a method reference or a lambda expression?

§17.9 Property Binding

16. The bind method is used for property binding. What type of interface does the bind method's parameter implement?

§17.10 JavaFX CSS

17. Provide a statement in which a label named heading calls setStyle to position itself top-center in its container and set the size of its font to 16.

18. What is the JavaFX CSS property that causes a horizontal line to be drawn through it, like this?

19. What is wrong with the following 45 degrees value for an -fx-rotate JavaFX CSS property?

```
-fx-rotate: 45 deg;
```

§17.11 Scene Graph Inheritance

20. If you specify style properties for a container, all of that container's properties automatically apply to all of that container's components. (T / F)

21. For a JavaFX CSS property that does not exhibit inheritance by default, what can you do to make it exhibit inheritance?

22. What does the .root selector do?

§17.12 Style Sheets and Cascading

23. Provide a statement that links a style sheet named petShop.css to a JavaFX window that has a Scene object named scene.

24. It is legal to add more than one style sheet file to a scene graph root container. (T / F)

25. With the higher priority places at the top, what is the cascade of places where JavaFX CSS property values can be assigned to a scene graph node?

§17.13 Button Control and FactorialButton Program

26. In the FactorialButton program, what statement prevents a user from entering anything directly into the factorial text box?

27. What JavaFX CSS property

determines the horizontal gap between components in a container?

§17.14 Distinguishing Between Multiple Events

28. Within an event handler, what is returned when there is an e.getSource() method call?

29. What Button method can you use to retrieve a button's label?

§17.15 Colors

30. Provide three style sheet rules with selectors named message1, message2, and message3 that use red, green, and blue to display an opaque dark magenta background color. For the three rules, use percentages for the first one, integers for the second one, and hexadecimal for the third one.

31. Provide a style sheet rule with a selector named label that uses an hsba construct to display a 50% transparent light blue background color.

Exercises

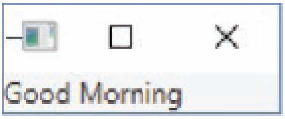
---

1. [after §17.1] In the first Java compiler, all GUI classes were bundled into one area of the Java API library and its name was the ____________.

2. [after §17.2] For each of the following, what Java API package must you import?

- **a)** Application
- **b)** Stage
- **c)** Scene
- **d)** Label

3. [after §17.2] Write a minimal JavaFX program that displays a Label containing “Good Morning” text. No need for comments and no need for a launch method. The display should look something like this:



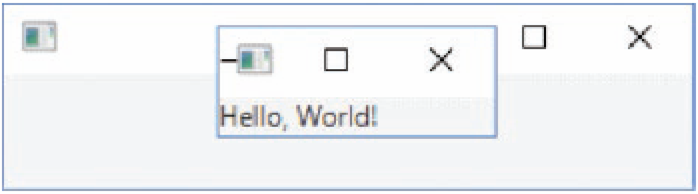
Good Morning

©JavaFX

4. [after §17.6] What can you do to prevent users from updating a TextField control named textBox?

5. [after §17.8] Give three examples of how a user might cause an event to be fired.

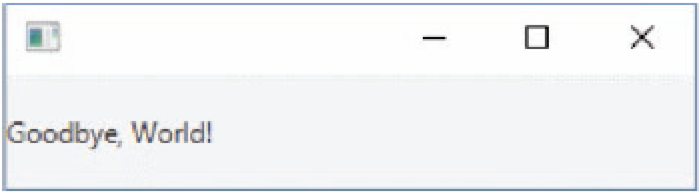
6. [after §17.8] A HelloGoodbye program generates two windows, like this:



Hello, World!

©JavaFX

After the user closes the smaller window by clicking its X, the outer window displays “Goodbye, World!”, like this:

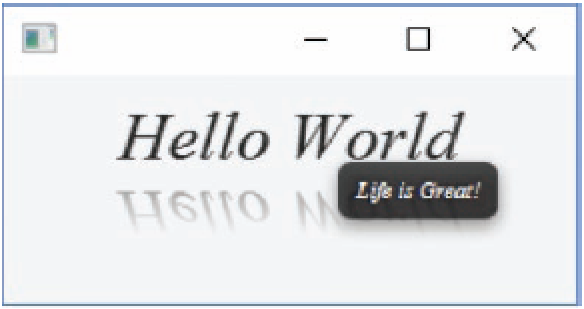

Goodbye, World!

©JavaFX

The setOnHiding method is similar to the setOnAction method in that it registers an event handler with a node in a scene graph. In the following skeleton code, replace *Insert* setOnHiding *method call here* with a call to setOnHiding, and in that call, use a lambda expression for setOnHiding’s argument.

```
public class HelloGoodbye extends Application
{
  private Label goodbye = new Label();

  public void start(Stage stage)
  {
    Stage secondaryStage = new Stage();

    stage.setScene(new Scene(goodbye, 300, 50));
    secondaryStage.setScene(new Scene(new Label("Hello, World!")));

    <Insert setOnHiding method call here for the secondaryStage window.>

    stage.show();
    secondaryStage.show();
  } // end start
} // end HelloGoodbye
```

7. [after §17.11] Complete the program that displays this "Hello World" message:



Hello World

Life is Great!

©JavaFX

Note these styling characteristics: (1) pane: top-center alignment with 10-point padding; (2) label's font: italic, 30 point, serif; (3) tooltip: 10-point font. Implement this styling by embedding setStyle method calls in this code fragment:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.*; // Label, Tooltip
import javafx.scene.effect.Reflection;
public class BigHello extends Application
{
  public void start(Stage stage)
  {
    Label label = new Label("Hello World");
    Tooltip tip = new Tooltip("Life is Great!");
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane, 250, 100);
    pane.getChildren().add(label);

    <Insert setStyle method call here.>

    label.setEffect(new Reflection());
    label.setTooltip(tip);
    stage.setScene(scene);
    stage.show();
  } // end start
} // end BigHello class
```

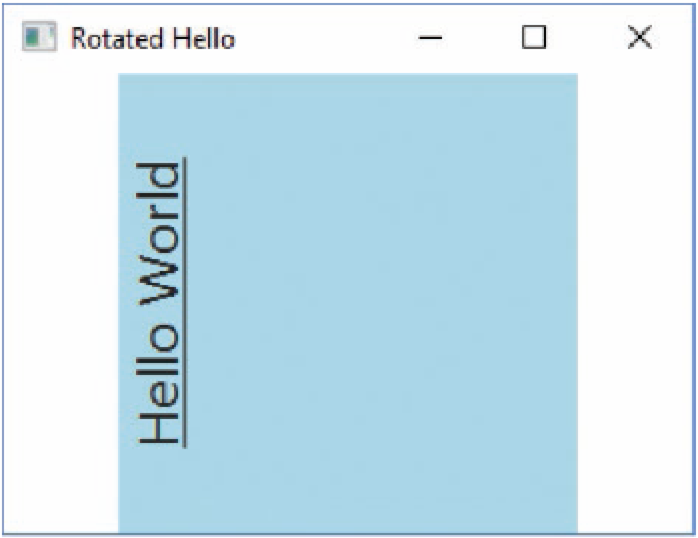
Page 846

8. [after §17.12] What are two advantages of using a style sheet as opposed to inline styles?

9. [after §17.12] Given this start method:

```
public void start(Stage stage)
{
  Label message = new Label("Hello World");
  Scene scene = new Scene(message, 300, 200);

  <style sheet registration statement goes here>

  stage.setTitle("Rotated Hello");
  stage.setScene(scene);
  stage.show();
} // end start
```

Using the above method, implement a program and a style sheet that, together, produce the following window. The width and height are 300 and 200 pixels, respectively, the color is light blue, and the font size is 24 point.



©JavaFX

10. [after §17.13] By calling setDisable(true), you can disable a button and give it a muted (grayed out) appearance and make its handler unresponsive to clicks on it. Modify the FactorialButton program so that the button is initially disabled. Enable it only after the user types a key when the xBox text box has focus. When the user clicks the Factorial button, calculate and display the factorial and disable the button.

Hints: (1) Declare btn to be an instance variable. (2) Instead of calling setOnAction, have your xBox control call setOnKeyTyped, like this:

```
xBox.setOnKeyTyped(e ->
btn.setDisable(false));
```

11. [after §17.15] What color

name is best for displaying something with a color reminiscent of a grass backyard?

12. [after §17.15] Provide three class selector rules for a style sheet, such that each rule displays an opaque dark yellow background color, but using a different technique. All three rules use an RGB value, but the first rule uses percentages, the second rule uses integers, and the third rule uses hexadecimal values. Use the names label1, label2, and label3 for the three selectors.

13. [after §17.15] Provide a class selector rule for a style sheet that uses an hsba construct to display a 50% transparent light green background color. Use the name header for the selector.

Review Question

Solutions

1. Swing components are considered to be “lightweight” because they are built with Java code and are “light” enough to “swing” from one platform to another.

2. RIA stands for Rich Internet Application. It’s a web application (a program that runs within a browser either on a PC or on a portable device) that is as fully functional as a typical desktop application.

3. JavaFX is considered to be an improvement over Swing and AWT because of its more consistent set of components, its built-in support for touch gestures, its formatting that matches the formatting found in web programming, and better support for animation.

4. Use the Application class as the superclass for your JavaFX programs.

5. The start method is special in that it's the starting point for all JavaFX programs, and it gets called automatically.

6. To cause a window to be a certain size, you can specify a scene's width and height in a Scene constructor, or you can call the Stage object's setWidth and setHeight methods.

7. True. You can position a program's window on the computer screen by calling the Stage object's setX and setY methods.

8. Label hello = new Label("Hello World!");

9. To limit a Label control to a width narrower than its surrounding container, you should call the setWrapText and setMaxWidth methods.

10. nameBox.setPrefColumnCount(nameBox
- 2);

11. The setPromptText method inserts grayed-out text into the text box to instruct user on what to enter. The text is not used as user input and is overlaid if and when the user enters something.

12. You can add multiple components to a scene by adding those components to a container (like a FlowPane object) and add the container to the scene as part of the scene’s constructor call.

13. An event is an object that tells your program that something has happened.

14. verifyButton.setOnAction(this::verify);

15. A method reference is better

for code reuse.

16. The bind method’s parameter implements the Observable interface.

17. heading.setStyle("-fx-alignment: top-center; -fx-font-size: 16");

18. -fx-strikethrough

19. The space is not allowed as part of the value. The value should be 45deg.

20. False. A container’s properties do not automatically apply to all of that container’s components.

21. To make a JavaFX CSS property exhibit inheritance, you can provide a property-value pair with inherit as the value. For example:

```
message.setStyle("-fx-text-fill:
inherit;");
```

22. .root is a selector that can be

used to apply property-value pairs to a program’s scene graph root.

```
scene.getStylesheets().add("petShop.css");
```

24. True. It is legal to add more than one style sheet file to a scene graph root container.

25. With the higher priority places at the top, here is the cascade of places where JavaFX CSS property values can be assigned to a scene graph node:

Inline styles

Parent style sheet

Scene style sheet

Component calls a styling method

User agent style sheet

26. xfBox.setEditable(false);

27. The -fx-hgap property determines the horizontal gap between components in a container.

28. The getSource method returns

a reference to the component whose event fired.

29. To retrieve a button's label, you can call the getText method.

30. .message1 {-fx-background-color: rgb(40%,0%,40%);}

.message2 {-fx-background-color: rgb(102,0,102);}

.message3 {-fx-background-color: #660066;}

31. .label {-fx-background-color: hsba(240,100%,90%,.5);}

---

[1] See optional GUI-track sections 1.10, 3.25, 5.9, Interlude, 8.16, 10.12, 11.10, 12.16, 14.11, 15.14, and 16.11.

[2] If you want to prevent window resizing, include the statement, stage.setResizable(false);

[3] The label's text is from *Hamlet*, ed. Barbara Mowat (New York: Simon & Schuster, 1992), 1.4.90. References are to act, scene, and line.

[4] Use of the term “lambda” (the Greek letter, λ) comes from the venerable programming language, Lisp. In Lisp, methods are referred to as functions, and functions that have no name (anonymous functions) are known as lambda functions. Likewise, Java’s lambda expressions are basically methods that have no name—thus the etymology of “lambda” for lambda expressions

[5] In many programming languages (including Java), there’s a method/function that returns the current time as the number of milliseconds past January 1, 1970. As the story goes, that date was chosen by Kernighan and Ritchie as the beginning of time for their progeny, UNIX and C. With its compactness and geeky etymology, hackers love K&R formatting, and refer to it as 1TBS (“the one true brace style”). For the Java source code in this book, we put the opening brace on its

own line because we think it leads to more readable code, but if you and your teacher prefer K&R for your Java source code, go for it!

[6] A *dictionary* was an ancient form of communication, used as a means to record word definitions. The definitions appeared on thin sheets of compressed wood fiber.

[7] The factorial of a number is the product of all positive integers less than or equal to the number. The factorial of n is written as n! Example: The factorial of 4 is written as 4!, and 4! is equal to 24 because 1 times 2 times 3 times 4 equals 24.

[8] In 1666, Isaac Newton discovered that white light is composed of all of the colors of the color spectrum. He showed that when white light is passed through a triangular prism, it separates into different colors. And when the resulting colors are passed through a

second triangular prism, they are brought back together to form the original white light.

GUI Programming—Layout Panes

Objectives

- Learn the benefits of using layout panes.
- Understand FlowPane layout details.
- Understand VBox and HBox

layout details.

- Understand BorderPane and GridPane layout details.
- Understand TilePane and TextFlow layout details.
- Learn how to embed layout panes inside other layout panes.

Outline

18.1 Introduction 849

18.2 Layout Panes 851

18.3 FlowPane and GridPane—Competing Layout Philosophies 853

18.4 VBox Program with Two Stages and an Image File 858

18.5 BorderPane 867

18.6 TilePane and TextFlow Containers 872

18.7 TicTacToe Program 878

18.8 Embedded Panes, HBox, and MathCalculator Program 882

18.9 Plain Pane Container and Component Positioning 889

18.1 Introduction

This is the second chapter in our three-chapter sequence on GUI programming. In the previous chapter, Chapter 17, you learned JavaFX GUI basics. You learned that a window's frame is a stage and what fills that frame is a scene. You learned that a scene always contains exactly one node—the root node of the scene graph. In the simplest cases, the scene's one node is a simple component, like a Label. In more complex cases, that one node is a container, like a FlowPane, which can hold more than one component.

You learned that a FlowPane arranges its components like a word processor arranges words on a page. It adds components left to right on the current row until it runs out of space on that row. Then it wraps around to the left side of the next row and continues from there. If the user changes the size of the window, the number of components on each row typically changes, and this can alter the layout in undesirable ways. We didn't admit this at the time, but to obtain satisfactory results, we needed to tinker around with different alignments and different values for scene width and height and pane padding and gaps.

The automatic layout adjustments in our prior examples were only somewhat helpful because (for simplicity) we used the same FlowPane container for every application. There are many other types of containers, and if you select the type of container judiciously, the default initial setting and/or the adjustments that occur automatically as users alter window size are beneficial. If you read the optional GUI sections at the ends of the earlier chapters, you will already be familiar with several of these other -containers.

We used the Group class in the GUI section programs in the Interlude and in Chapters 5, 11, 13, and 14. The Group class automatically positions added components on top of each other in the upper left corner of the screen, unless the programmer explicitly specifies the components' locations. Although the programmer usually needs to specify each component's location, Group automatically conforms its container size (enlarging or shrinking) to include all its components—wherever they are and however large they may be. The Group class is intended for applications where the programmer wants to position components explicitly.

If you want that explicit positioning but do not want the container's size to conform to what's inside of it, there is another option—the Pane class. We'll describe the Pane class at the end of this chapter.

We used the FlowPane, SplitPane, and VBox container classes in the optional GUI section program in Chapter 8. A SplitPane differs from a FlowPane in that there's no wrapping and there are visible user-movable dividers between adjacent components. A VBox differs from a FlowPane in that there's no wrapping and it's vertically oriented. Later in this chapter, we'll provide a detailed description of VBox and another illustration of its use, and you'll see other examples of VBox in subsequent sections.

We used the StackPane class in the GUI section programs in the Interlude and in Chapter 13. A StackPane is like a VBox, but instead of adding new components below previous components (in the y-direction), new components are added on top of previous components (i.e., they "stack" on top of each other). Although you can specify other component positions, by default all components are centered in the StackPane container, and they stay centered if the user changes window dimensions. A StackPane makes it easy to paste a relatively small label on the center of a larger image behind it. If a subsequent component is somewhat transparent, the previous component will show through, facilitating creation of interesting artistic effects.

In this chapter, we'll start with an

overview of basic layout principles for designing GUI windows. We’ll then introduce you to several containers – GridPane, VBox, BorderPane, TilePane, TextFlow, and HBox.– and provide examples that illustrate how they can be used for different layout purposes. We’ll show how you can embed these containers inside other containers to further refine a window’s layout.

Figure 18.1 includes almost[1] all of the JavaFX classes used in this book. In the figure, the classes with bold borders are the ones that are used in this chapter and the next chapter. As you study this chapter and the next one, you may want to refer back to this figure from time to time. It can help you recall inherited methods that may be useful.

In Figure 18.1, note the Pane class and its children. These are container classes, and they are the focus of this chapter. Also note the classes under the abstract Control class. These are the classes for objects that are known as controls. You learned about some of them in the previous chapter, and you'll learn about more of them in the next chapter. The programs in the next chapter that introduce those additional controls will provide additional practice with this chapter's containers.

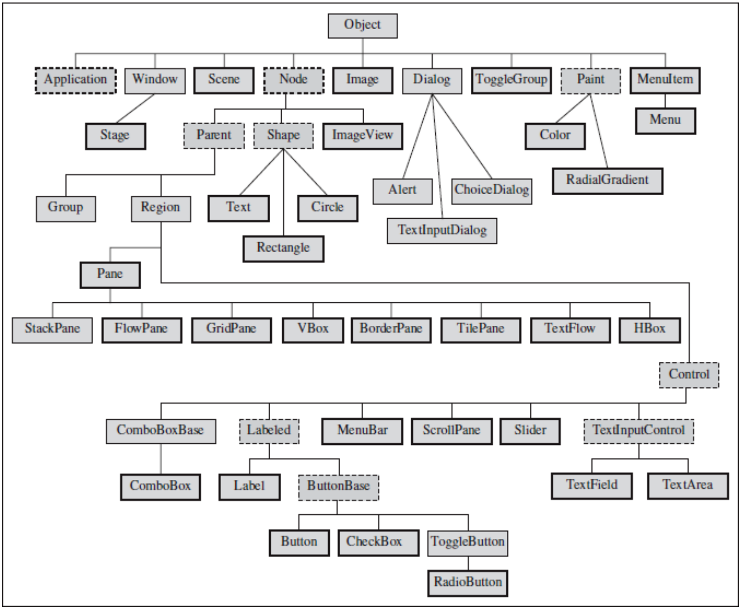

Object
Application
Window
Scene
Node
Image
Dialog
ToggleGroup
Paint
MenuItem
Menu
Stage
Parent
Shape
ImageView
Color
RadialGradient
Alert
ChoiceDialog
TextInputDialog
Group
Region
Text
Circle
Rectangle
Pane
StackPane
FlowPane
GridPane
VBox
BorderPane
TilePane
TextFlow
HBox
Control
ComboBoxBase
Labeled
MenuBar
ScrollPane
Slider
TextInputControl
ComboBox
Label
ButtonBase
TextField
TextArea
Button
CheckBox
ToggleButton
RadioButton

Figure 18.1 Inheritance relationships among JavaFX classes

The classes with bold borders appear in examples in Chapters 18 and 19. The classes with dashed borders are abstract classes.

18.2 Layout Panes

The interface for text-based programs is pretty straightforward—just ask the user a question and wait for the user to provide an answer. With GUI programs, the interface involves much more work on the part of the programmer, but the result is worth it, right? The user gets a fun, colorful window with multiple points of interaction. Yay! To make that interface easy to understand, it's important that you, the programmer, position the window's components appropriately.

In the old days, positioning components was a tedious, manual process. Programmers would spend hours calculating and devising formulas for the space needed for each component and the pixel coordinate positions for each component, as functions of window size. Today, programmers are freed from that tedium because layout panes do all that calculation work automatically in the background.

Page 853

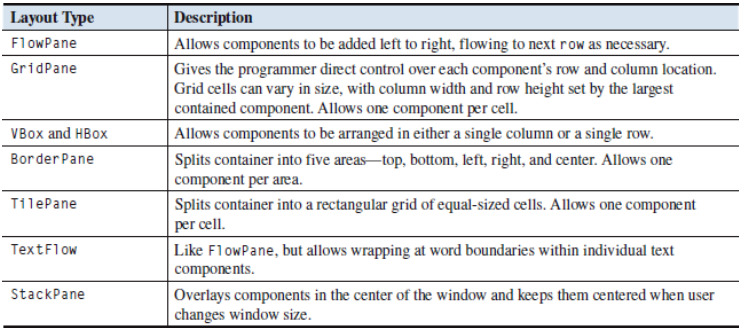
A *layout pane* is a container class whose purpose is to manage the positioning of components that have been added to the container. In the previous chapter, our layout pane of choice was the FlowPane. In upcoming GUI programs, we'll use a wide range of layout panes. In general, each layout pane's goal is to arrange components neatly and adjust that arrangement appropriately if the user resizes the window. Usually, the neatness goal equates to making sure components are aligned and making sure components are appropriately spaced within the pane.

If a user adjusts a window's size, or

if an event handler adjusts a component's size or position, the Java Virtual Machine (JVM) recalculates the pixel coordinate positions for all of the components in each of the window's layout panes.

As mentioned earlier, there are different types of layout panes, and they employ different strategies for positioning their components. The table in Figure 18.2 describes those strategies for the more popular layout pane classes. You should already be familiar with the table's first entry, FlowPane. We'll cover the other layout panes later in this chapter, except for the StackPane container, which was covered in GUI sections earlier in the book.

Figure 18.2's layout pane classes are provided in the javafx.scene.layout package. Except for the TextFlow class, which is provided in the javafx.scene.text package. It might seem odd that TextFlow, VBox, and HBox are referred to as layout panes even though they don't have "pane" in their class names. As an alternative, you can use the more generic term "layout container" instead of layout pane, but we'll stick with "layout pane."



Layout Type	Description
`FlowPane`	Allows components to be added left to right, flowing to next `row` as necessary.
`GridPane`	Gives the programmer direct control over each component's row and column location. Grid cells can vary in size, with column width and row height set by the largest contained component. Allows one component per cell.
`VBox` and `HBox`	Allows components to be arranged in either a single column or a single row.
`BorderPane`	Splits container into five areas—top, bottom, left, right, and center. Allows one component per area.
`TilePane`	Splits container into a rectangular grid of equal-sized cells. Allows one component per cell.
`TextFlow`	Like `FlowPane`, but allows wrapping at word boundaries within individual text components.
`StackPane`	Overlays components in the center of the window and keeps them centered when user changes window size.

Figure 18.2 Several of the more popular layout pane classes

To assign a layout pane to a window, use a pane argument when instantiating a Scene. For example:

```
FlowPane pane = new FlowPane();
Scene scene = new Scene(pane, WIDTH, HEIGHT);
```

For the layout panes other than

FlowPane, you'll sometimes have arguments in their constructor calls, and the arguments will be different for the different layout panes. We'll get to those details later on.

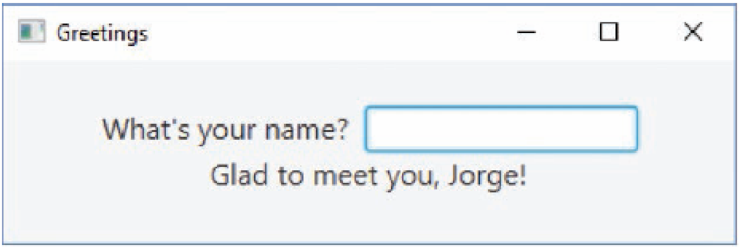
All of the classes in Figure 18.2 are descendants of the Pane class, which is in the javafx.scene.layout package. The Pane class automatically positions components on top of each other in its top-left corner. It's legal to use the Pane class to hold your window's components, but it's not as useful as its subclass containers and therefore it's not used all that much. But the Pane class has lots of useful methods that its subclasses inherit. Pane and its descendants are resizable alternatives to the Group class, which we used extensively in end-of-chapter GUI sections. A Group automatically collapses itself to just barely contain all of its components, whereas a Pane automatically expands itself to fill the container that surrounds it.

18.3 FlowPane and GridPane—Competing Layout Philosophies

In the previous chapter, we wanted to present GUI basics without getting bogged down in layout details, so we used the simple FlowPane container, which doesn't require much explanation. Although easy to use, the FlowPane container can be frustrating because you don't have much control over what it does. If you need more direct control over where things go, you should consider using the GridPane container. This section illustrates the trade-offs in choosing between the FlowPane and GridPane containers.

FlowPane Layout Mechanism

As you learned earlier, the FlowPane container implements a simple one-compartment layout scheme that accepts multiple components. By default, when the first component gets added to the compartment, it gets positioned in the top-left corner. Subsequent components get positioned at the right of previously added components. If there is not enough room to add a component at the right of previously added components, the new component is placed on the next line. That is, it “flows” to the left side of the next line. Now let’s go back and revisit the previous chapter’s Greeting program, described in Section 17.7. That program prompts the user to enter his or her name and prints a personalized greeting after the user presses enter. Here’s what the program displays after the user types Jorge and presses Enter:

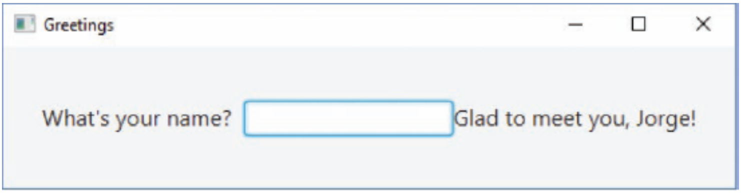


What's your name?

Glad to meet you, Jorge!

©JavaFX

And here's what happens after the user expands the window by dragging its right border to the right:

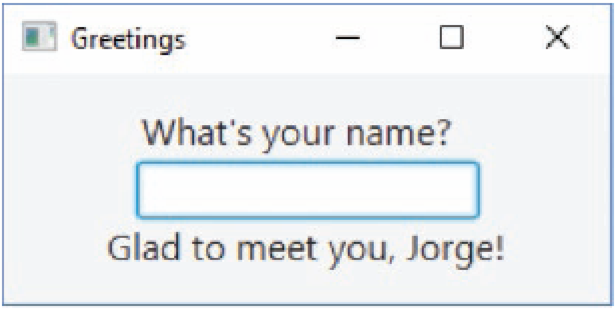


What's your name?

Glad to meet you, Jorge!

©JavaFX

Kinda ugly, eh? And if the user shrinks the window by dragging its right border to the left, here's the result:



What's your name?

Glad to meet you, Jorge!

©JavaFX

Note how the components are centered both horizontally and vertically. That’s because our program calls setAlignment like this:

```
pane.setAlignment(Pos.CENTER);
```

As an alternative, if you’d like to use JavaFX CSS for the alignment, you could do this:

```
pane.setStyle("-fx-alignment: center;");
```

And if you’d like a different alignment, you could use one of these other values for the -fx-alignment property:

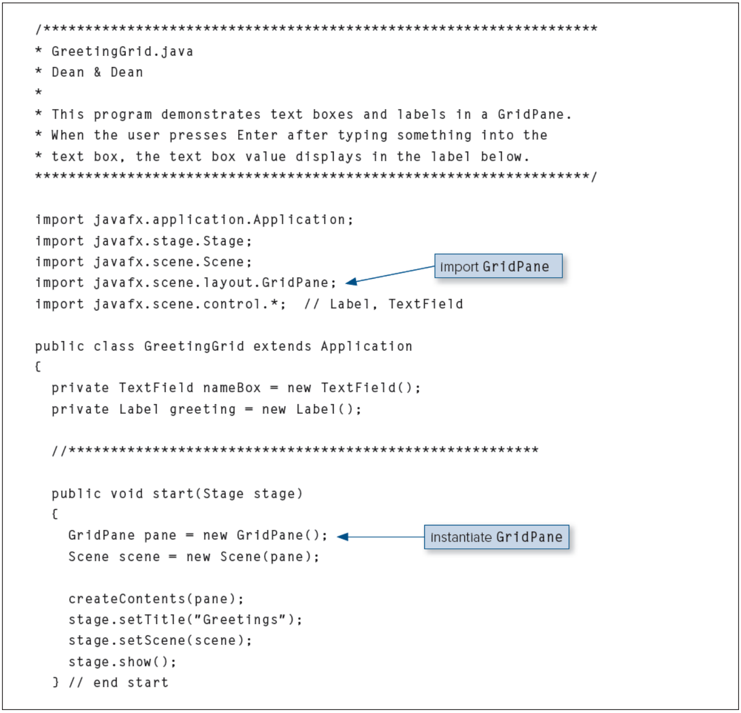
top-left, top-center, top-right, center-left, center-right, bottom-left, bottom-center, bottom-right

GridPane Layout

With FlowPane layouts, components move around as the window’s size and the components’ sizes change. That can be disconcerting. If you know enough about your data sizes and you make the window large enough to show everything you want, you can stabilize your windows by switching to a different layout—the layout produced by a GridPane. A GridPane splits the container into a rectangular grid, and you explicitly specify the column and row for each component. This keeps components from changing column and row positions.

Let's use the GridPane to convert the last chapter's Greeting program, so it has a more stable layout. See the GreetingGrid program in Figures 18.3a and 18.3b. Note how much of its code is the same as that in the original Greeting program. The differences are that we instantiate and use a GridPane object instead of a FlowPane object, we use three add method calls to add the three components to the pane, we use inline styles for formatting, and we use a lambda expression for the event handler. We'll focus on the GridPane issues.

Page 856



```
/*********************************************************************
* GreetingGrid.java
* Dean & Dean
*
* This program demonstrates text boxes and labels in a GridPane.
* When the user presses Enter after typing something into the
* text box, the text box value displays in the label below.
*********************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.GridPane;
import javafx.scene.control.*;  // Label, TextField

public class GreetingGrid extends Application
{
  private TextField nameBox = new TextField();
  private Label greeting = new Label();

  //**********************************************************

  public void start(Stage stage)
  {
    GridPane pane = new GridPane();
    Scene scene = new Scene(pane);

    createContents(pane);
    stage.setTitle("Greetings");
    stage.setScene(scene);
    stage.show();
  } // end start
```

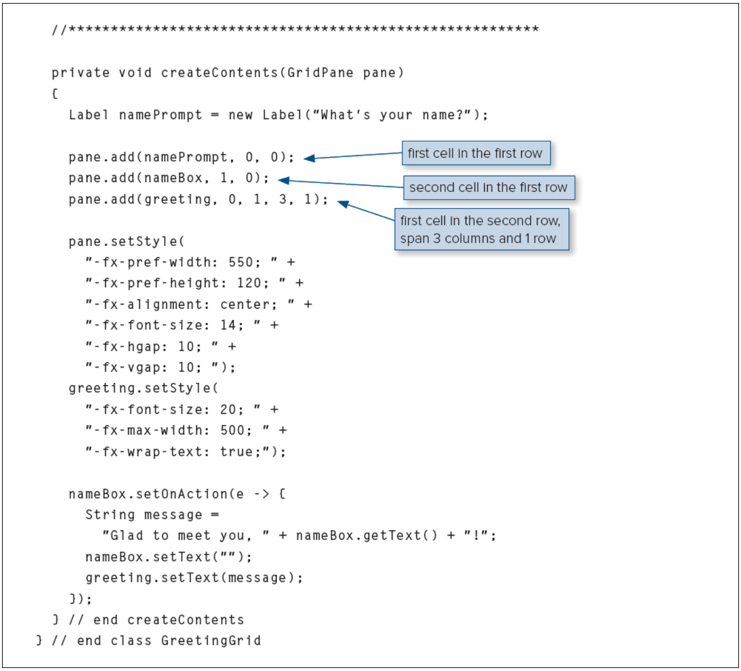
Figure 18.3a GreetingGrid program—part A

You might recall that in the original Greeting program, we called the getChildren and addAll methods to add all three components to the FlowPane. Here's the code that was used:

```
pane.getChildren().addAll(namePrompt, nameBox, greeting);
```

With the GreetingGrid program, we replace that line with these three lines:

```
pane.add(namePrompt, 0, 0);
pane.add(nameBox, 1, 0);
pane.add(greeting, 0, 1, 3, 1);
```



```
  //********************************************************

  private void createContents(GridPane pane)
  {
    Label namePrompt = new Label("What's your name?");

    pane.add(namePrompt, 0, 0);
    pane.add(nameBox, 1, 0);
    pane.add(greeting, 0, 1, 3, 1);

    pane.setStyle(
      "-fx-pref-width: 550; " +
      "-fx-pref-height: 120; " +
      "-fx-alignment: center; " +
      "-fx-font-size: 14; " +
      "-fx-hgap: 10; " +
      "-fx-vgap: 10; ");
    greeting.setStyle(
      "-fx-font-size: 20; " +
      "-fx-max-width: 500; " +
      "-fx-wrap-text: true;");

    nameBox.setOnAction(e -> {
      String message =
        "Glad to meet you, " + nameBox.getText() + "!";
      nameBox.setText("");
      greeting.setText(message);
    });
  } // end createContents
} // end class GreetingGrid
```

first cell in the first row

second cell in the first row

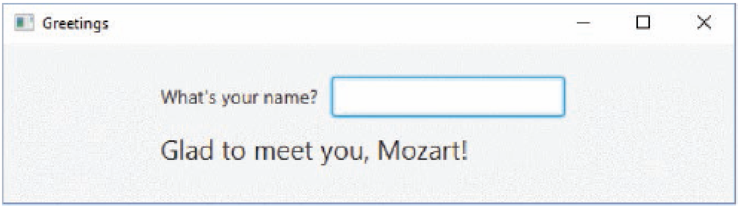
first cell in the second row, span 3 columns and 1 row

Figure 18.3b GreetingGrid program—part B

Instead of arranging the three components as arguments within an addAll method call, now we have three add method calls, each one in charge of positioning one component in the grid. By the way, you could change the order of the three method calls, and it wouldn’t matter. The add method call’s first argument is the component. The second argument is the index of the grid column where you want that component to go, with 0 being the first column. The third argument is the index of the grid row where you want that component to go, with 0 being the first row. Recall that with a two-dimensional array, you specify row, then column. Try to remember that this argument sequence is different—column, then row.

The add method’s fourth and fifth arguments are optional, and they specify the number of columns and rows, respectively, that can be used by the component, if the component needs to use more than one cell. Thus, the greeting component goes into the cell at column 0, row 1, and it has three columns and one row allocated to it. In other words, the greeting component may use all the space it needs in the first, second, and third columns in the second row.

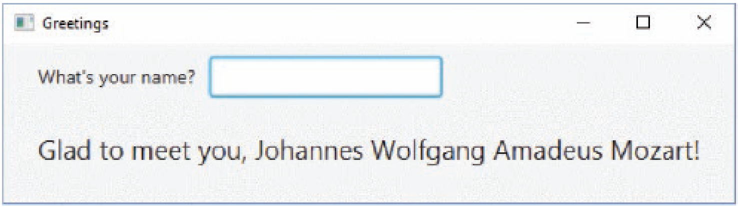
Here is what the GreetingGrid program produces with Mozart's last name entered and then his full name entered:



What's your name?

Glad to meet you, Mozart!

©JavaFX

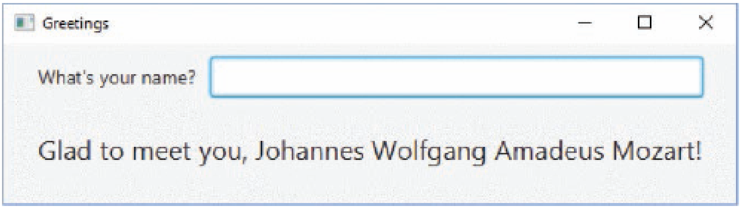


What's your name?

Glad to meet you, Johannes Wolfgang Amadeus Mozart!

©JavaFX

The GridPane layout's first two column widths are set by the default widths of the name prompt and the name box, respectively. You can see that the greeting label at the bottom takes all the space in both these columns plus more if necessary. Because we allocated three columns for the greeting, the third column becomes wide enough to handle the greeting overflow. If we had allocated just two columns to the greeting, the second column would have expanded to handle the long name. That would be appropriate for the long greeting, but it would introduce a problem for the name box. Unless you explicitly specify a component's size, a component will expand to fill the size of its containing cell. So with a long greeting assigned to just the first two columns (and not three), the name box would expand, with its right edge matching the long greeting's right edge. Here's the result, with an unusually wide name box:



What's your name?

Glad to meet you, Johannes Wolfgang Amadeus Mozart!

©JavaFX

If you don't want to rely on a GridPane's components to determine the GridPane's column widths and row heights, there is an alternative. Starting at column zero, you can specify successive column widths explicitly, with statements like these:

```
pane.getColumnConstraints().add(new ColumnConstraints(125));
pane.getColumnConstraints().add(new ColumnConstraints(150));
```

The ColumnConstraints class is in the javafx.scene.layout package. These two statements force the first column to be 125 pixels wide and the second column to be 150 pixels wide, regardless of the sizes of components in those columns. Using a zero-argument ColumnConstraints constructor call lets the GridPane's internal layout manager, rather than the programmer, determine how wide to make a particular column. You can do a similar thing with rows. Just substitute Row for Column in the above statements.

18.4 VBox Program with Two Stages and an Image File

In this section, we'll implement a program that uses two different stages: One stage uses a FlowPane to retrieve user input, and one stage uses a VBox pane to display output labels that are positioned vertically (the V in VBox stands for vertical). By the way, there's also an HBox pane that we'll use later that positions components horizontally (the H in HBox stands for horizontal).

Example—A Dance Recital Poster

Suppose you want to implement a program that prints a poster with details about an upcoming dance recital. You want your program to be flexible, so it can be reused later for other performances. Thus, rather than hard coding the event details, the program prompts the user for event details, and then displays those details in a pleasing format as a poster. Specifically, the poster should display these items, one above the other:

- The name of the performance.
- A promotional image, like a photograph of the featured performer.
- The performance date.
- The performance time.
- The performance location.

The five items' values come from the five user inputs shown in Figure 18.4.

Each input uses the same text box control, with the text box control getting a different prompt for each new input (courtesy of an event handler that updates a label that serves as the text box's prompt). Figure 18.5 shows the program's response after the user enters the last input, for the performance's venue.

Figures 18.6a through 18.6c show the Recital program's code. Figure 18.6a shows the program's import statements and instance variables. The data instance variable is an array of strings for the five input items. The index instance variable keeps track of the index position within the data array for the data that is being processed.

Figure 18.6b contains the start method. It begins with initialization statements for variables that support the input gathering portion of the program. The first such initialization assigns the different prompt strings to a prompts array.

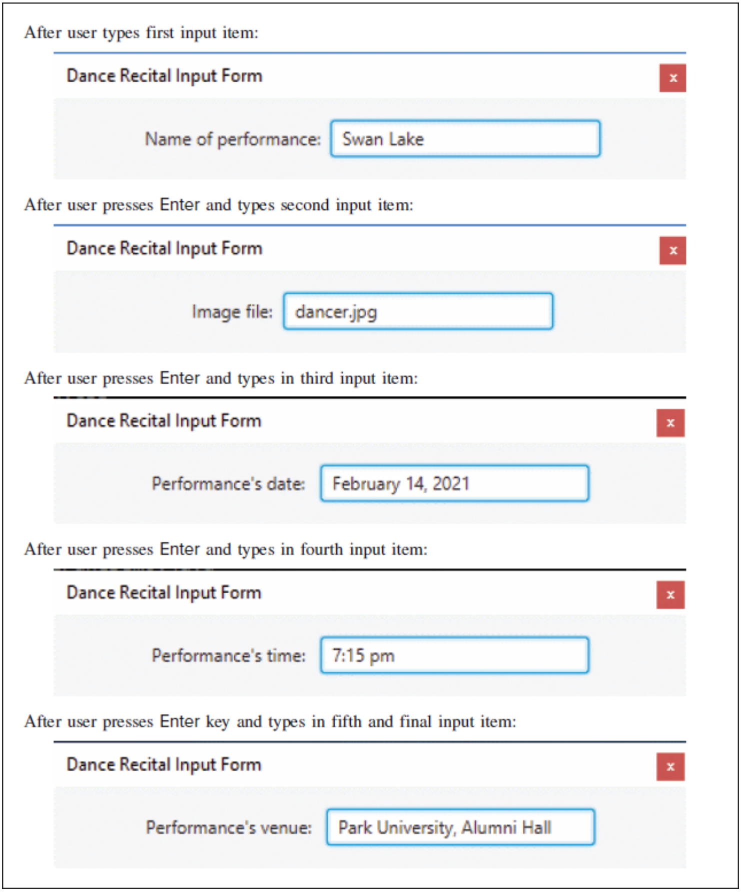
The fourth initialization assigns an instantiated FlowPane like this:

FlowPane pane = new FlowPane(prompt, dataBox);

In the past, we've called the FlowPane constructor with empty parentheses. By calling the constructor with component arguments, there's no need to call the add method later on. The two components get added in the manner you'd expect—centered with prompt first and dataBox next.

After the initializations, we call the stage's initStyle method with the StageStyle.UTILITY argument. That generates a no-frills stage that omits the minimize and maximize buttons in the title bar (see Figure 18.4 to verify the omissions). The input gathering process is rather mundane, and the no-frills stage fits the task nicely. The next two statements connect the recitalInput.css style sheet and its .dataBox rule to the program. More on the CSS stuff a bit later.

Page 860



After user types first input item:

Dance Recital Input Form
×

Name of performance: Swan Lake

After user presses Enter and types second input item:

Dance Recital Input Form
×

Image file: dancer.jpg

After user presses Enter and types in third input item:

Dance Recital Input Form
×

Performance's date: February 14, 2021

After user presses Enter and types in fourth input item:

Dance Recital Input Form
×

Performance's time: 7:15 pm

After user presses Enter key and types in fifth and final input item:

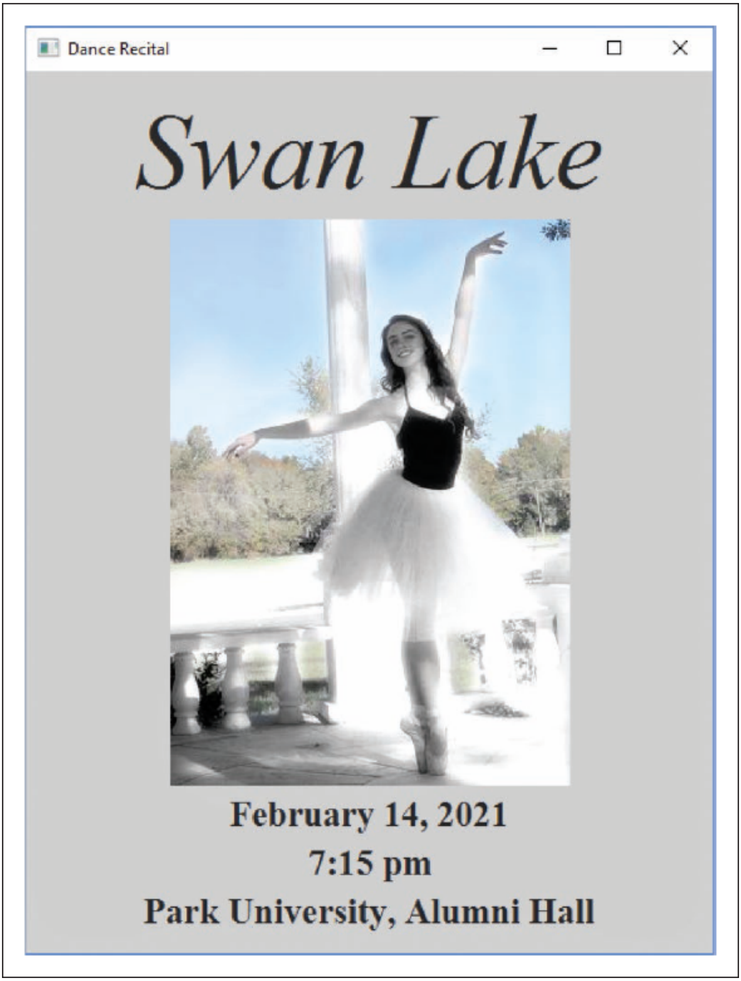
Dance Recital Input Form
×

Performance's venue: Park University, Alumni Hall

Figure 18.4 Sample session for the Recital program

©JavaFX

Page 861



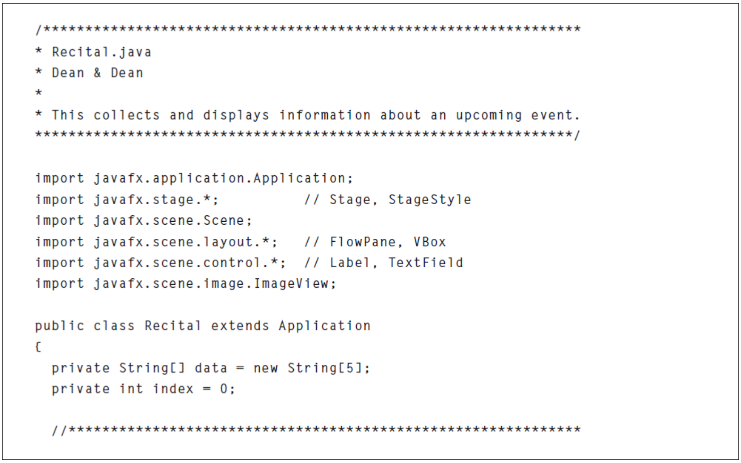
Swan Lake

February 14, 2021
7:15 pm
Park University, Alumni Hall

Figure 18.5 Representative poster created by the Recital program

The pictured ballerina, Jessica Sapenaro, is a Park University computer science alumna, a software contractor, and a professional dancer.

©JavaFX; Courtesy of Shane Immelt



```
/*******************************************************************
* Recital.java
* Dean & Dean
*
* This collects and displays information about an upcoming event.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.*;          // Stage, StageStyle
import javafx.scene.Scene;
import javafx.scene.layout.*;   // FlowPane, VBox
import javafx.scene.control.*;  // Label, TextField
import javafx.scene.image.ImageView;

public class Recital extends Application
{
  private String[] data = new String[5];
  private int index = 0;

  //****************************************************************
```

Figure 18.6a Recital program—part A

Next in the start method are the usual calls to setTitle, setScene, and show to assign the title, assign the scene, and make the stage visible, respectively. Next, the text box control calls setOnAction, which registers a lambda expression to handle what happens when the user presses Enter after typing text into the text box. Whenever this event occurs, the computer copies the user's entry into the program's data array at the current index position. Then it increments index in preparation for the next user entry. If this next index value is less than data.length, the program updates the prompt label with the index--positioned string from the prompts array. But if the next index value is beyond the range of the prompts and data arrays, the program closes the stage and calls the displayPoster method to display the dance recital's information within a new window.

Let's now examine the declarations for the variables that the lambda expression uses. We declare the data array to be an instance variable because it's used in two different methods (start and displayPoster). We declare dataBox, prompt, and prompts as local variables within the start method because they are used only in the start method and no other methods. The index variable is also used only in the start method, but we declare it as an instance variable. Why? Because it needs to persist for multiple executions of the lambda expression. If index were to be declared as a local variable in start, the compiler would generate an error message “local variables referenced from a lambda expression must be final or effectively final.” What in the world does that mean? The local variables dataBox, prompt, and prompt are reference variables, and their reference values don’t change, so they are considered final by the compiler. All that changes are the objects that the references point to. On the other hand, index is a primitive variable, and its value does change—thus the compilation error about it not being effectively final.

```
// Generate a window that gathers input for the event by
// displaying a text box and a prompt that changes for each input.

public void start(Stage stage)
{
  String[] prompts = {"Name of performance:",
    "Image file:", "Performance's date:",
    "Performance's time:", "Performance's venue:"};
  Label prompt = new Label(prompts[0]);
  TextField dataBox = new TextField();
  FlowPane pane = new FlowPane(prompt, dataBox);
  Scene scene = new Scene(pane);

  stage.initStyle(StageStyle.UTILITY);
  scene.getStylesheets().add("recitalInput.css");
  dataBox.getStyleClass().add("dataBox");
  stage.setTitle("Dance Recital Input Form");
  stage.setScene(scene);
  stage.show();

  dataBox.setOnAction(e -> {
    data[index] = dataBox.getText();
    dataBox.setText("");
    index++;
    if (index < data.length)
    {
      prompt.setText(prompts[index]);
    }
    else
    {
      stage.close();
      displayPoster();
    }
  });
} // end start
```

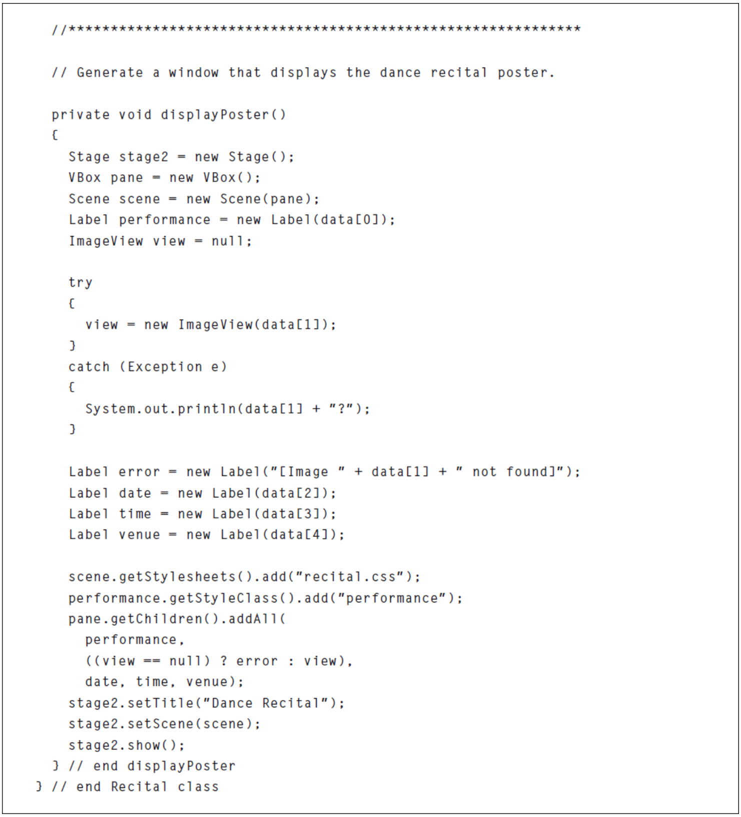
You can call the `FlowPane` constructor with components, and those components get added to the pane.

This populates the `data` array with user inputs.

This displays the dance recital's information within a new window.

Figure 18.6b Recital program—part B

Let's now examine the displayPoster method in Figure 18.6c. It starts off by generating a new stage, which serves as a repository for displaying the dance recital's information. We're using a new stage because the original stage was no-frills (with no minimize and maximize buttons), and we don't want that this time; for the dance recital "poster," we want a standard window, in all its glory. After instantiating the new stage, the method instantiates VBox and Scene objects, designating the VBox container as the root of the stage's scene graph. A VBox is like a vertically oriented FlowPane in that you simply add components without specifying their locations. The difference is that the components get added using a top-to-bottom layout scheme, with only one column no matter how many components.



```
  //*************************************************************

  // Generate a window that displays the dance recital poster.

  private void displayPoster()
  {
    Stage stage2 = new Stage();
    VBox pane = new VBox();
    Scene scene = new Scene(pane);
    Label performance = new Label(data[0]);
    ImageView view = null;

    try
    {
      view = new ImageView(data[1]);
    }
    catch (Exception e)
    {
      System.out.println(data[1] + ”?”);
    }

    Label error = new Label(”[Image ” + data[1] + ” not found]”);
    Label date = new Label(data[2]);
    Label time = new Label(data[3]);
    Label venue = new Label(data[4]);

    scene.getStylesheets().add(”recital.css”);
    performance.getStyleClass().add(”performance”);
    pane.getChildren().addAll(
      performance,
      ((view == null) ? error : view),
      date, time, venue);
    stage2.setTitle(”Dance Recital”);
    stage2.setScene(scene);
    stage2.show();
  } // end displayPoster
} // end Recital class
```

Figure 18.6c Recital program—part C

By default, there’s no spacing/gap between a VBox’s added components. If you want to introduce such spacing, you can call the VBox constructor with an argument that represents the spacing in pixels between the VBox container’s components. For example:

VBox pane = new VBox(5);

As an alternative, you can introduce spacing with the -fx-spacing property in a style sheet, which is what you'll see in the Recital program's style sheet, coming up.

In the displayPoster method, note this ImageView code:

```
ImageView view = null;

try
{
  view = new ImageView(data[1]);
}
catch (Exception e)
{
  System.out.println(data[1] + "?");
}
```

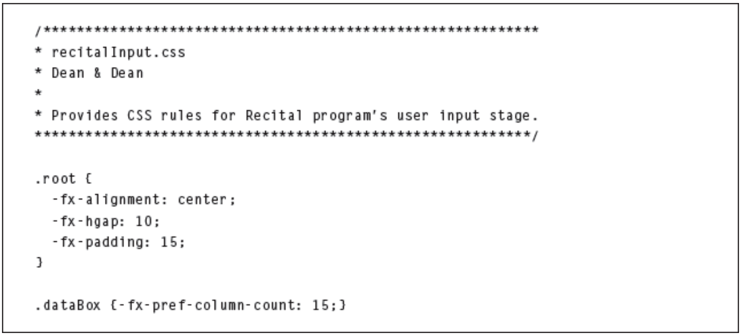
An ImageView object encapsulates an image file so the file's image can be displayed in a scene graph like other components. Above, we initialize the ImageView reference variable, view, with a default value of null. The subsequent statement tries to replace the null value with an image, whose file is stored at the location specified by the string in data[1]. If that location does not match an actual image file in the current directory, the JVM throws an exception, and the catch block prints the image filename entered by the user to help the user identify a possible input error. Because System.out.println is used, that error message appears in the console window.

Continuing with our examination of the displayPoster method, the next four lines create labels to hold an error message and information about the dance recital. Next up, we call getStylesheets and getStyleClass to connect the recital.css style sheet and its .performance rule to the VBox pane and the performance label, respectively. Then we add all the components to the scene graph. For the addAll method call's second argument, note how we use a conditional operator to determine the component to be added. If view is still null, that means there was a problem instantiating the ImageView object, so we use the error label. Otherwise, we use the ImageView object to display the image. The displayPoster method's last three statements assign the stage's title, assign its scene, and make the stage visible.

Now let's dig into the style sheets for the program's two stages. Figure 18.7a shows the style sheet for the first stage, the one for user inputs. Note how the .root rule centers the FlowPane container's components and adds a horizontal gap between the components. Note also how the .root rule applies padding to the scene graph root, the FlowPane container. Without that padding, the window's top and bottom borders would touch the prompt and text box—how unsightly! Why is the padding necessary for this program, but not our prior programs? With our prior programs, we specified the window's dimensions as part of the Scene instantiation or with JavaFX CSS rules. This time, we rely on the contents to determine the window size, so we need padding to provide spacing at the top and bottom. At the bottom of the style sheet, you can see the .dataBox rule. It uses the -fx-pref-column-count property to ensure that the text box has plenty of room for the user's inputs.

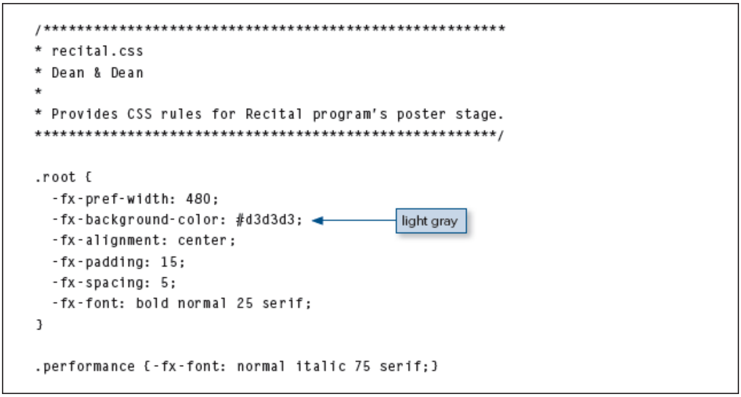
Figure 18.7b shows the style sheet for the second stage, the one for the dance recital poster. Note how the .root rule applies preferred width, background color, alignment, padding, spacing, and font values to the scene graph root, the VBox container. We've used all of those properties in prior examples except for the -fx-spacing and -fx-font properties. The -fx-spacing property works only with the VBox and

HBox panes, and as its name suggests, it introduces spacing between the container's added components. The recital.css style sheet assigns 5 to the -fx-spacing property, which generates 5 pixels of space between every component in the VBox container. Note how the -fx-font property has four values assigned to it. The -fx-font property is known as a *shorthand property* in that it's a compact way to take care of a group of related properties. Specifically, in using the -fx-font property, you specify values for each of the -fx-font-weight, -fx-font-style, -fx-font-size, and -fx-font-family properties. So the following line generates a boldface, nonitalicized, 25-point, serif font:



```
/*************************************************************
* recitalInput.css
* Dean & Dean
*
* Provides CSS rules for Recital program's user input stage.
*************************************************************/

.root {
  -fx-alignment: center;
  -fx-hgap: 10;
  -fx-padding: 15;
}

.dataBox {-fx-pref-column-count: 15;}
```

Figure 18.7a recitalInput.css style sheet



```
/*********************************************************
* recital.css
* Dean & Dean
*
* Provides CSS rules for Recital program's poster stage.
*********************************************************/

.root {
  -fx-pref-width: 480;
  -fx-background-color: #d3d3d3;
  -fx-alignment: center;
  -fx-padding: 15;
  -fx-spacing: 5;
  -fx-font: bold normal 25 serif;
}

.performance {-fx-font: normal italic 75 serif;}
```

light gray

Figure 18.7b recital.css style sheet

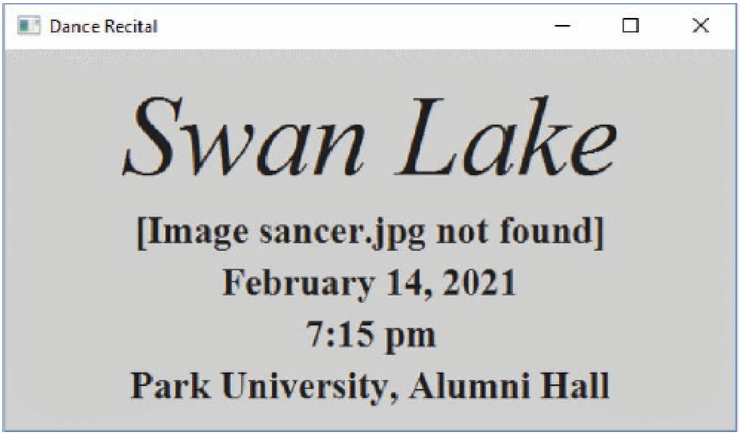
-fx-font: bold normal 25 serif;

Note how the style sheet's .performance rule also uses the -fx-font shorthand property to specify several font values. Specifically, it generates a normal weight, italicized, 75-point, serif font.

Let's return to the issue of what happens when the user enters an invalid filename for the image file. The program allows the user to continue entering input values because at this point they are all just strings. After the user makes the last entry, the invalid filename displays on the console followed by a question mark. If the user enters sancer.jpg, then the program would display this on the console:

```
>sancer.jpg?
```

And the program would generate this GUI window:



Swan Lake

[Image sancer.jpg not found]

February 14, 2021

7:15 pm

Park University, Alumni Hall

©JavaFX

Note how the VBox height adapts automatically to the height of its contents. You may have noticed that this section's VBox description was not as long as previous descriptions of FlowPane or GridPane. That's because VBox doesn't need much attention. It just arranges components from top to bottom in order of entry by calling getChildren().add or getChildren().addAll. Easy peasy, lemon squeezy!

One issue that requires a bit of attention is if you have a TextField control (or a TextArea control, which we'll get to in the next chapter) in a VBox pane. The default is for the TextField to span the entire width of the pane, even if you call setPrefColumnCount with a value less than the pane's width. Ouch. But not to worry—just toggle the fillWidth property to off by calling setFillWidth like this:

```
pane.setFillWidth(false);
```

18.5 BorderPane

Page 868

FlowPane and VBox layouts are nice because they allow you to enter components one after another and let the computer handle the details. GridPane is nice because it lets you explicitly position the components. But sometimes you want something that's more sophisticated than FlowPane or VBox and more automatic than GridPane. You may want a BorderPane.

BorderPane Areas

A BorderPane is particularly useful for windows that need components near their edges. It's common to put a title at the top of a window. It's common to put a menu at the left of a window. It's common to put buttons at the bottom of a window. The BorderPane accommodates all those situations by splitting its container into five layout areas. Four of the areas are near the edges and one is in the center. As you'll see later when we get to the code, those areas are referred to with the names center, top, right, bottom, and left. Note the areas' positions in Figure 18.8. The dashed lines don't appear on the actual display. We've drawn them in to show you the area boundaries.

The sizes of the five areas are determined at runtime, and they're based on the contents of each area. Thus, if the left area contains a wide component, such as a label with a lot of text, the JVM widens the left area. But if the left area contains a narrow component, the JVM narrows the left area.

If an outer area is empty, it collapses

so that it does not take up any space. But what exactly happens during the collapse? Each outer area controls only one dividing line, so only one dividing line moves for each collapsed area. Figure 18.8 shows you that the left area's dividing line is the boundary between left and center, the top area's dividing line is the boundary between top and below, and so on. Thus if the top area is empty, the top dividing line moves all the way up to the top border, and the left, center, and right areas all expand upward. What happens if the right and bottom areas are both empty? The right area being empty causes the right dividing line to move all the way to the right border. The bottom area being empty causes the bottom dividing line to move all the way down to the bottom border. Here's the resulting layout:

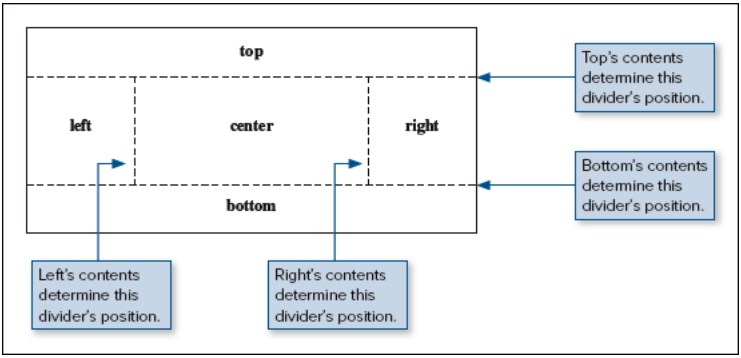
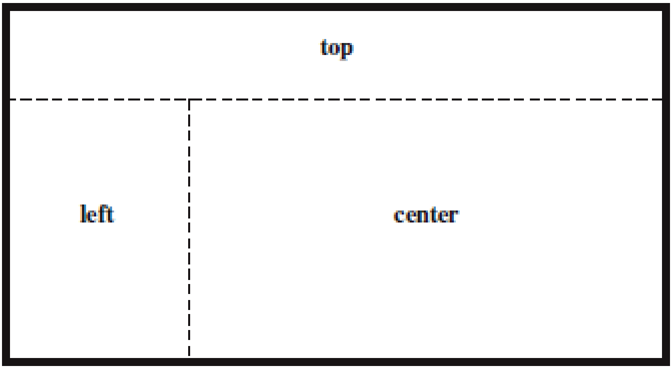

top
left
center
right
bottom
Top's contents determine this divider's position.
Bottom's contents determine this divider's position.
Left's contents determine this divider's position.
Right's contents determine this divider's position.

Figure 18.8 BorderPane areas


top
left
center

Once again, the dashed lines don't appear on the actual window. What happens if the center area is empty? The center area doesn't control any of the dividing lines, so nothing happens.

Adding Components

Creating a BorderPane and adding

components to it is fairly straightforward. Here’s an example that calls setTop to add a “Welcome to Your Life” label to the top area of a BorderPane:

```
BorderPane pane = new BorderPane();
pane.setTop(new Label("Welcome to Your Life"));
```

As you’d probably guess, there are comparable methods (setCenter, setRight, setBottom, and setLeft) that add components to the other four areas.

In addition to the zero-parameter BorderPane constructor shown above, there is also a one-parameter constructor whose parameter is for the center component, and there is a five-parameter constructor whose parameters are for the components that get stored in the pane’s five compartments. The five passed-in components must be in the order center, top, right, bottom, and left.

With a FlowPane, you can add as many components as you like. With a BorderPane, you can add only five components directly, one for each of the five areas. If you add a component to an area that already has a component, then the new component replaces the old component. Thus, in executing the following statements, the “You’re such a big mess.” label replaces the “We’re back in business.” label:

```
pane.setCenter(new Label("We're back in business."));
pane.setCenter(new Label("You're such a big mess."));
```

⚠ If you want to add more than one component to an area, it’s easy to make the mistake of assigning to the same area twice. After all, there’s no compilation error to warn you of your misdeed. But what you really need to do is add a pane to where you would like to have multiple components. Then put the multiple components into that added pane. We'll discuss this pane-in-a-pane strategy later in the chapter.

AfricanCountries Program

Let's put this BorderPane stuff into practice by using it within a complete program. In our AfricanCountries program, we add buttons with African country names to the five areas of a BorderPane container. Figure 18.9 shows the program's output. The reason we're using buttons instead of labels is because we want to make the area borders more obvious, and buttons have built-in borders and labels do not. To further highlight the area borders, we've added code that expands the buttons, so each button fills its compartment. Also, we've added code that creates margins around each of the five buttons and padding inside the BorderPane's perimeter.

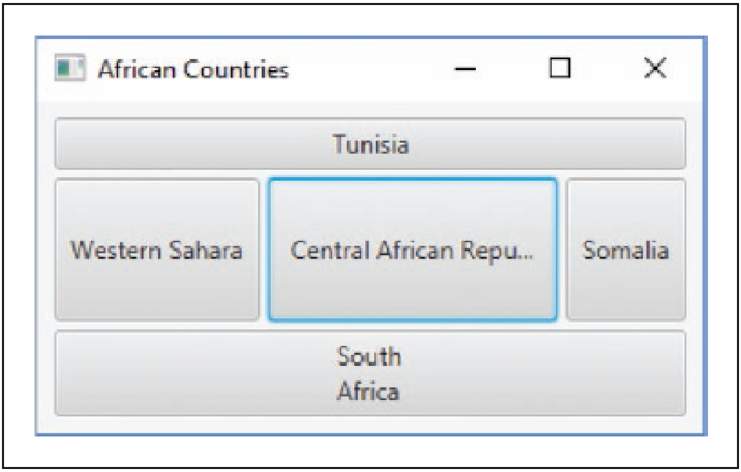

African Countries
Tunisia
Western Sahara
Central African Repu...
Somalia
South
Africa

Figure 18.9 Resulting window for the AfricanCountries program

©JavaFX

In the AfricanCountries window, note how the left and right buttons are wide enough to display their labels fully, whereas the center button is too narrow to display its label fully. Remember that the two vertical dividing lines (separating the left, center, and right areas) accommodate the contents in the left and right areas, and the center area gets what's left over. Note that the bottom button has a label that spans two lines. The button is tall enough to display its two-line label fully because the bottom dividing line accommodates the bottom area's contents.

Skim through the AfricanCountries program listing in Figure 18.10. Most of the code is straightforward, but some of it deserves a closer look. Note the setCenter, setTop, setRight, setBottom, and setLeft method calls to add the five buttons to the BorderPane's five areas. And for the setBottom method call, note the \n between "South" and "Africa." That's the reason that the button's label spans two lines.

Also in the AfricanCountries program, note this statement:

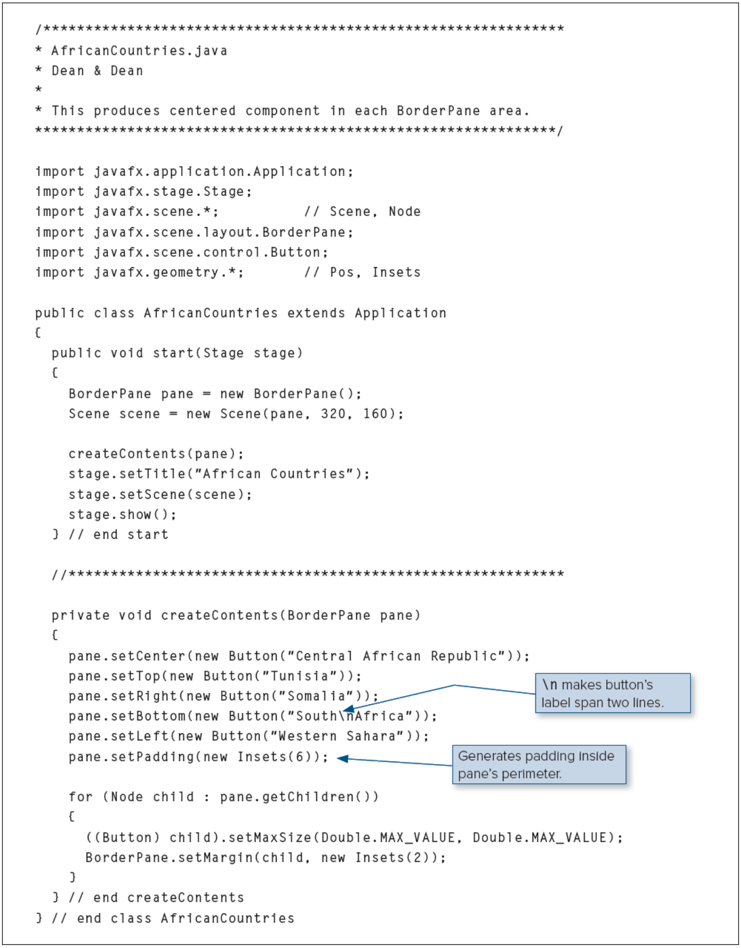
```
pane.setPadding(new Insets(6));
```

It generates 6 pixels of padding inside the BorderPane's perimeter. The Insets constructor call creates space for the four sides of a rectangle. In this case, the space rectangle gets applied to the inside of a BorderPane. In the one-parameter Insets constructor call shown here, the parameter's value specifies the same spacing for all four sides of the rectangle. There's also a four-parameter Insets constructor, whose parameters specify the spacing for the top, right, bottom, and left sides, in that order.

Here's the code that expands the buttons and adds margins around each of them:

```
for (Node child : pane.getChildren())
{
  ((Button) child).setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
  BorderPane.setMargin(child, new Insets(2));
}
```

The getChildren method retrieves the pane container's child nodes. In this case, that means the five buttons. The Button class's setMaxSize method expands each button so it fills its compartment. The setMargin method call assigns a 2-pixel margin around each button. That means there's a 4-pixel gap between adjacent buttons (2 pixels from each button's margin).



```
/*******************************************************************
* AfricanCountries.java
* Dean & Dean
*
* This produces centered component in each BorderPane area.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.*;          // Scene, Node
import javafx.scene.layout.BorderPane;
import javafx.scene.control.Button;
import javafx.geometry.*;       // Pos, Insets

public class AfricanCountries extends Application
{
  public void start(Stage stage)
  {
    BorderPane pane = new BorderPane();
    Scene scene = new Scene(pane, 320, 160);

    createContents(pane);
    stage.setTitle("African Countries");
    stage.setScene(scene);
    stage.show();
  } // end start

  //***************************************************************

  private void createContents(BorderPane pane)
  {
    pane.setCenter(new Button("Central African Republic"));
    pane.setTop(new Button("Tunisia"));
    pane.setRight(new Button("Somalia"));
    pane.setBottom(new Button("South\nAfrica"));
    pane.setLeft(new Button("Western Sahara"));
    pane.setPadding(new Insets(6));

    for (Node child : pane.getChildren())
    {
      ((Button) child).setMaxSize(Double.MAX_VALUE, Double.MAX_VALUE);
      BorderPane.setMargin(child, new Insets(2));
    }
  } // end createContents
} // end class AfricanCountries
```

\n makes button's label span two lines.

Generates padding inside pane's perimeter.

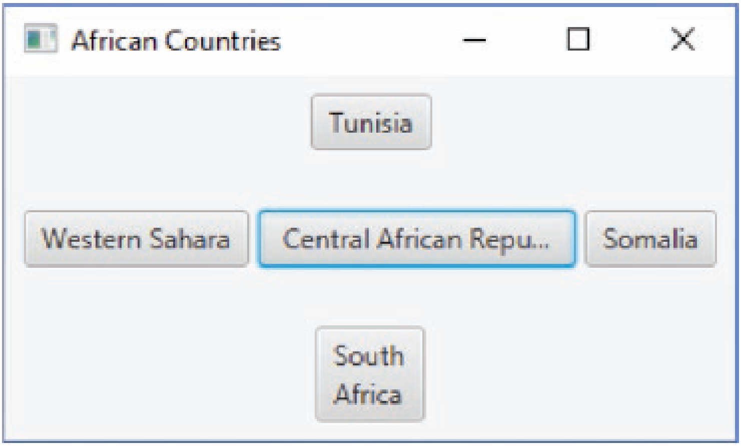
Figure 18.10 AfricanCountries program

Now let's modify the

AfricanCountries program so its buttons are their natural sizes. That means getting rid of the setMaxSize method call and adding a center alignment method call. Here is the program's modified for-each loop:

```
for (Node child : pane.getChildren())
{
  BorderPane.setMargin(child, new Insets(2));
  BorderPane.setAlignment(child, Pos.CENTER);
}
```

With those changes in place, the window looks like this:

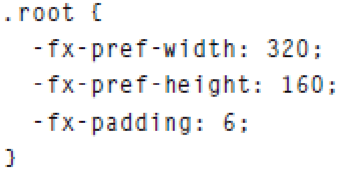

African Countries
Tunisia
Western Sahara
Central African Repu...
Somalia
South Africa

©JavaFX

Without the buttons' explicit center alignment, each button would use its BorderPane area's default alignment—

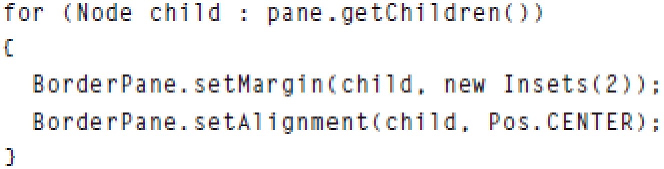
left alignment for the top and bottom areas, top alignment for the left and right areas, and center alignment for the center area. For one of the end-of-chapter exercises, you’re asked to show what the window looks like when there’s no setAlignment method call.

With the trend toward using style sheets for styling, you might be asking why we didn’t use a style sheet for the AfricanCountries program. Well, we could have used a style sheet to define the window’s dimensions and its padding. Specifically, this rule would do the trick:



```
.root {
  -fx-pref-width: 320;
  -fx-pref-height: 160;
  -fx-padding: 6;
}
```

But for styling the buttons, unfortunately there are no JavaFX CSS margin and alignment properties that work for the Button control. That’s why we are forced to use BorderPane’s static styling methods, setAlignment and setMargin. Here’s that code once again:



```
for (Node child : pane.getChildren())
{
  BorderPane.setMargin(child, new Insets(2));
  BorderPane.setAlignment(child, Pos.CENTER);
}
```

Above, you can see that each component in the BorderPane gets the same styling. Suppose you want to provide customized styling for the components. To do that, you can have pane call getCenter(), getTop(), getRight(), getBottom(), or getLeft() to retrieve the component in the area that you’re interested in and then use the retrieved component as an argument when calling one of the styling methods. For example:

BorderPane.setAlignment(pane.getCenter()
Pos.TOP_LEFT);

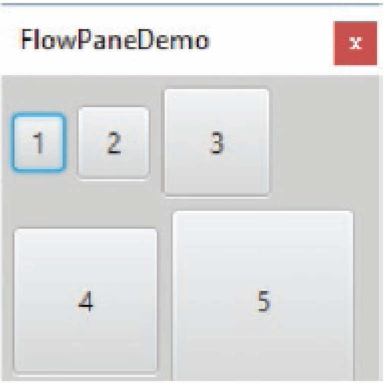
18.6 TilePane and TextFlow Containers

The BorderPane's partitioning scheme (top, left, center, right, and bottom) works well for situations that fit into its pattern. A GridPane enables you to put components at specific places in a rectangular grid. It gives you precise control over where things are relative to each other – independent of window size.

A FlowPane adds components at the right of the previously added component. If the component bumps into the window's right border, it gets positioned at the left side of the next row. If components are not all the same size, those in subsequent rows do not align in columns below - components above them, and in general the lengths of different rows are not the same. For example, note the window at the right, which comes from a program (FlowPaneDemo) that uses the FlowPane container.

See the the FlowPaneDemo program

in Figure 18.11. As the scene.getStylesheets().add("flow.css"); statement indicates, the program uses a style sheet, whose code is shown in Figure 18.12. We'll use that same style sheet with upcoming programs to illustrate use of the TilePane and TextFlow containers. All three programs will use the .root class selector rule. But FlowPaneDemo is the only one that uses the .flowPane rule, TilePaneDemo is the only one that uses the .tilePane rule, and TextFlowDemo is the only one that uses the .love rule.

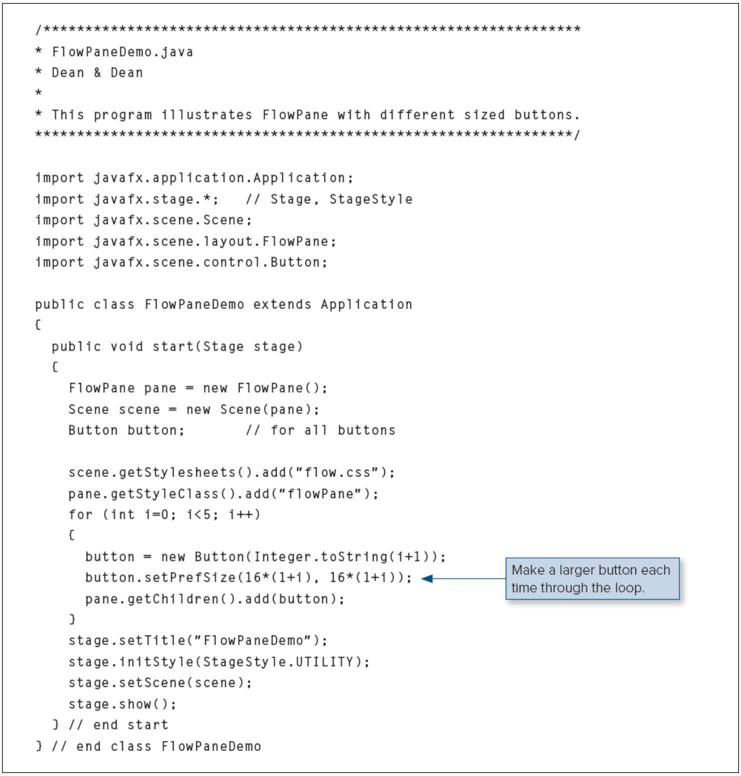


©JavaFX

Note the program's for loop. It instantiates five buttons and adds them to the pane. Note the program's stage.initStyle(StageStyle.UTILITY); method call. As you learned in the prior chapter, the stageStyle.UTILITY value gets rid of the minimize and maximize icons in the title bar. Go back to the program's generated window, and you can see how the simplified title bar is necessary so there's sufficient room for the window's title.

TilePane

A TilePane is like a FlowPane in that it adds components at the right of the previously added component. If the component bumps into the pane's right border, it gets positioned at the left side of the next row. In adding a component to a TilePane, the JVM puts it in a box (a "tile"), whose width matches the width of the widest component in the TilePane and whose height matches the height of the tallest component in the TilePane. All the components in a TilePane use a tile with that same size. Remember—TilePane containers use uniform-sized tiles.

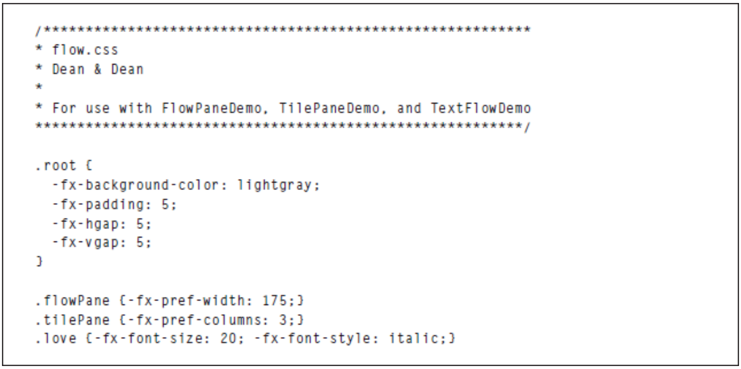


```
/*******************************************************************
* FlowPaneDemo.java
* Dean & Dean
*
* This program illustrates FlowPane with different sized buttons.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.*;   // Stage, StageStyle
import javafx.scene.Scene;
import javafx.scene.layout.FlowPane;
import javafx.scene.control.Button;

public class FlowPaneDemo extends Application
{
  public void start(Stage stage)
  {
    FlowPane pane = new FlowPane();
    Scene scene = new Scene(pane);
    Button button;       // for all buttons

    scene.getStylesheets().add("flow.css");
    pane.getStyleClass().add("flowPane");
    for (int i=0; i<5; i++)
    {
      button = new Button(Integer.toString(i+1));
      button.setPrefSize(16*(1+i), 16*(1+i));
      pane.getChildren().add(button);
    }
    stage.setTitle("FlowPaneDemo");
    stage.initStyle(StageStyle.UTILITY);
    stage.setScene(scene);
    stage.show();
  } // end start
} // end class FlowPaneDemo
```

Make a larger button each time through the loop.

Figure 18.11 FlowPaneDemo program supported by CSS file in Figure 18.12

The default width for a TilePane is five columns, where each column is the width of one tile. The number of columns determines the pane’s dimensions: (1) The pane’s width is the number of columns times the width of each tile, (2) The pane’s height is the number of rows necessary to hold the components’ tiles times the height of each tile. Normally, you’ll want to control a TilePane window’s dimensions explicitly. To do so, you can specify the pixel dimensions in the Scene constructor call (as you’ve seen in many prior examples). But with a TilePane, as an alternative, you can establish the pane’s dimensions with the help of an -fx-pref-columns property in a style sheet. For example, we use the following rule in the flow.css style sheet for the TilePaneDemo program:

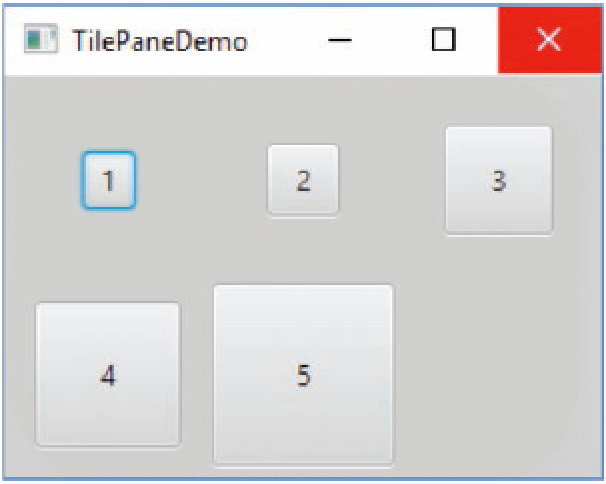


```
/**********************************************************
* flow.css
* Dean & Dean
*
* For use with FlowPaneDemo, TilePaneDemo, and TextFlowDemo
**********************************************************/

.root {
  -fx-background-color: lightgray;
  -fx-padding: 5;
  -fx-hgap: 5;
  -fx-vgap: 5;
}

.flowPane {-fx-pref-width: 175;}
.tilePane {-fx-pref-columns: 3;}
.love {-fx-font-size: 20; -fx-font-style: italic;}
```

Figure 18.12 Shared style sheet for FlowPaneDemo, TilePaneDemo, and TextFlowDemo programs

    -fx-pref-columns: 3;

That rule causes the window to have three columns. As explained earlier, the number of columns (working in conjunction with the number of tiles and the size of each tile) determines the pane's dimensions.



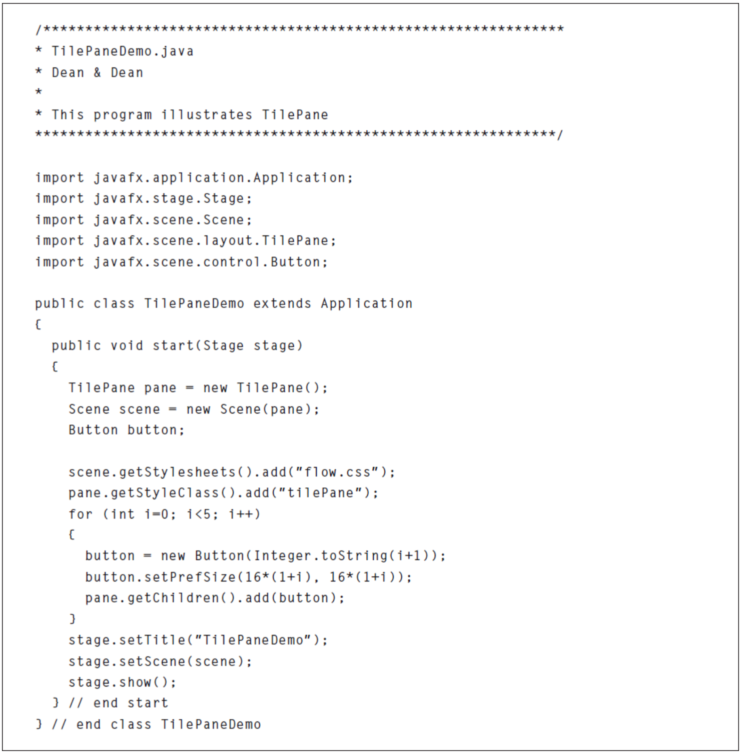
©JavaFX

See the TilePaneDemo program in Figure 18.13. It’s identical to the FlowPaneDemo program, except that it uses a TilePane, and it uses the .tilePane rule shown in Figure 18.12’s style sheet. That rule specifies 3 for the preferred number of columns. The image at the right shows the output for the TilePaneDemo program, and you can see that it does indeed have three columns. The first two columns hold two tiles each and the last column holds one tile. The tile boundaries are invisible, but you should be able to imagine where they are. The fifth button is the largest, so it determines the size of all five tiles. The other buttons are centered within their tiles, with no gaps between the tiles.

In the TilePaneDemo program, components are added to the TilePane with a horizontal layout scheme—that means left-to-right before going to the next row. As an alternative, you can specify a vertical layout scheme (top to bottom in the first column before going to the next column) by calling the one-parameter TilePane constructor with a value of Orientation.VERTICAL. For example:

```
pane = new TilePane(Orientation.VERTICAL)
```

The Orientation class is in the javafx.geometry package, so you'll need to import that package.



```
/****************************************************************
* TilePaneDemo.java
* Dean & Dean
*
* This program illustrates TilePane
****************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.TilePane;
import javafx.scene.control.Button;

public class TilePaneDemo extends Application
{
  public void start(Stage stage)
  {
    TilePane pane = new TilePane();
    Scene scene = new Scene(pane);
    Button button;

    scene.getStylesheets().add("flow.css");
    pane.getStyleClass().add("tilePane");
    for (int i=0; i<5; i++)
    {
      button = new Button(Integer.toString(i+1));
      button.setPrefSize(16*(1+i), 16*(1+i));
      pane.getChildren().add(button);
    }
    stage.setTitle("TilePaneDemo");
    stage.setScene(scene);
    stage.show();
  } // end start
} // end class TilePaneDemo
```

Figure 18.13 TilePaneDemo program

By default, the TilePane displays its tiles with no gaps between them. The two-parameter TilePane constructor allows you to introduce gaps. For example, here’s how to instantiate a TilePane with a 2-pixel horizontal gap and a 4-pixel vertical gap:

pane = new TilePane(2, 4);

There's also a three-parameter TilePane constructor that allows you to specify both vertical orientation and gaps. For example:

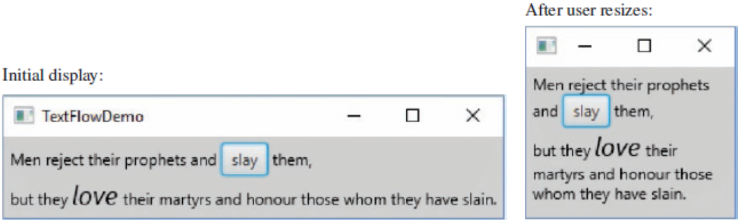
```
pane = new TilePane(Orientation.VERTICAL, 2, 4);
```

TextFlow

Suppose you want a container that displays text like a word processor, wrapping to the next line when there's an enter keystroke and when a word bumps into the document's right edge. But you also want it to accept insertion of arbitrary Node components between words anywhere in the text. You could put each word in a separate Label and stream them one after another with the other nodes in a horizontal FlowPane. But if most of what you have is ordinary text, this procedure would require instantiation of an awful lot of labels. Ugh! A better way is to put contiguous strings of text in Text objects, and add those Text objects along with the other nodes in a TextFlow container.

In the simplest case, where all you want is a block of uniformly formatted text, you can put one big string into a single Text component and add that component to the TextFlow container. In addition to wrapping at \n characters, the JVM would recognize spaces in the Text component's string and generate a line wrap at the end of any word that bumps into the right edge of the TextFlow container. That's like the behavior of a Label component that calls setWraptext(true). The more interesting scenario is when a TextFlow container has multiple components of varying types and styles. The line wrap works the same as before, but now it works not only for the words in the TextFlow container, but also any other types of components added to the container, such as buttons or image objects.[2]

The screenshot on the left below[3] is from the TextFlowDemo program, which adds text components and a button to a TextFlow container. In the screenshot on the right below, note how the words wrap when they bump into the window's right edge.[4]



Initial display:

Men reject their prophets and them,

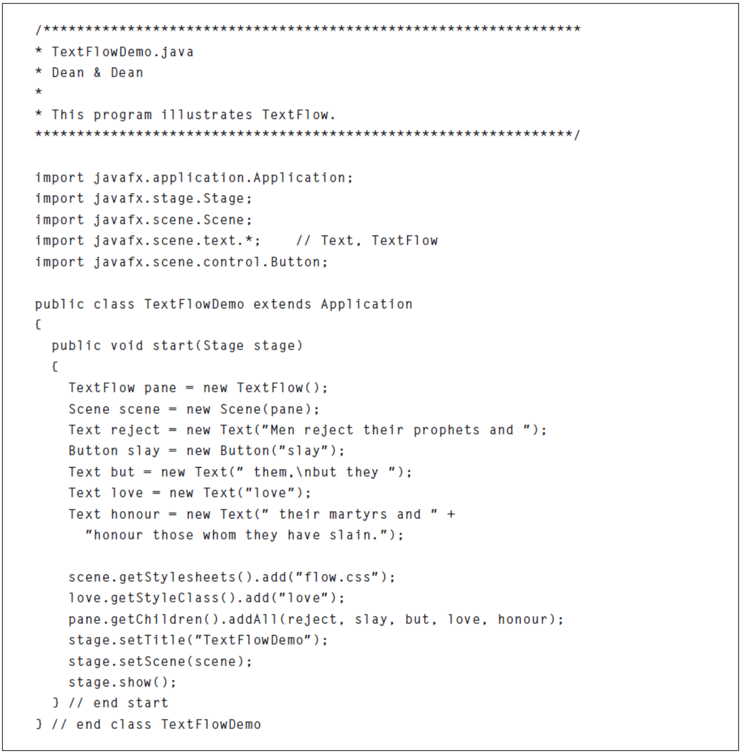
but they *love* their martyrs and honour those whom they have slain.

After user resizes:

Men reject their prophets
and them,
but they *love* their
martyrs and honour those
whom they have slain.

©JavaFX

Take a look at the TextFlowDemo program in Figure 18.14. TextFlow is in the javafx.scene.text package, so we have to import that package rather than the javafx.scene.layout package as with our prior programs. In the start method, note the TextFlow constructor call. Next up are the instantiations for four Text components and one Button component. We call the addAll method to add a Text component, the button, and then the three other Text components. If we had wanted all the text to have the same style, we could have combined the last three Text components into one Text component. Having three separate components allows us to specify a distinctive large italic font for the middle component (the component that says “love”). In the generated window, note how, despite its larger font size, the word “love” still fits naturally into the continuous stream of text. And likewise, the button also fits naturally into the continuous stream of text. Note how there’s a new line after the word “them,”. That’s from the \n character in the second Text component.



```
/*******************************************************************
* TextFlowDemo.java
* Dean & Dean
*
* This program illustrates TextFlow.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.text.*;    // Text, TextFlow
import javafx.scene.control.Button;

public class TextFlowDemo extends Application
{
  public void start(Stage stage)
  {
    TextFlow pane = new TextFlow();
    Scene scene = new Scene(pane);
    Text reject = new Text(”Men reject their prophets and ”);
    Button slay = new Button(”slay”);
    Text but = new Text(” them,\nbut they ”);
    Text love = new Text(”love”);
    Text honour = new Text(” their martyrs and ” +
      ”honour those whom they have slain.”);

    scene.getStylesheets().add(”flow.css”);
    love.getStyleClass().add(”love”);
    pane.getChildren().addAll(reject, slay, but, love, honour);
    stage.setTitle(”TextFlowDemo”);
    stage.setScene(scene);
    stage.show();
  } // end start
} // end class TextFlowDemo
```

Figure 18.14 TextFlowDemo program

In the middle of the program, note the scene.getStylesheets().add("flow.css") call, which loads the program's style sheet. Referring back to the style sheet's code in Figure 18.12, you can see that the .root rule applies a light gray color to the window's background. The .root rule's padding and gap property-value pairs are ignored because the TextFlow pane doesn't support those properties. In other words, you can't add padding inside the perimeter of a TextFlow container, and you can't add gaps between its nodes. The style sheet's .love rule applies an italicized 20-point font to the love Text component.

18.7 TicTacToe Program

In this section, we develop a tic-tac-toe program. The three-row by three-column board nicely illustrates some of the trade-offs among various types of containers.

User Interface

The program initially displays a three-row, three-column grid of blank buttons. Two users, player X and player O, take turns clicking blank buttons. Player X goes first. When player X clicks a button, the button's label changes from blank to X. When player O clicks a button, the button's label changes from blank to O. Player X wins by getting 3 X's in a row, 3 X's in a column, or 3 X's along a diagonal. Player O wins in the same manner, but with O's instead of X's. See the sample session in Figure 18.15.

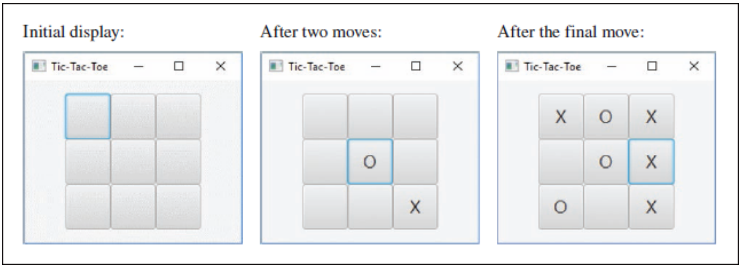

Initial display:
After two moves:
After the final move:
Tic-Tac-Toe
Tic-Tac-Toe
Tic-Tac-Toe
O
X
X
O
X
O
X
O
X

Figure 18.15 Sample session for the TicTacToe program

©JavaFX

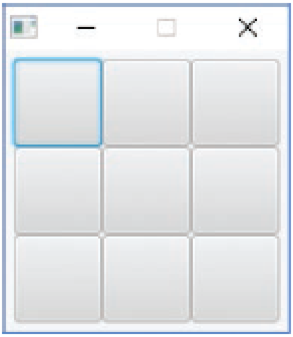
Program Design

The natural size of a button is just large enough to hold the button's label with a bit of padding surrounding it. For a tic-tac-toe board, natural-size buttons would be too small for a pleasing interface. Thus, for the TicTacToe program, we expand the buttons, with the help of a style sheet, so each one has a uniform size.

The buttons need to be added to a layout pane, with three buttons per row. The simplest layout pane is the FlowPane. For the FlowPane to work, you'd need to provide a width for the pane such that it's wide enough to hold three buttons in a row, but not four.

Having to calculate the pane's width from the width of three buttons in a row is rather inelegant, so let's try another pane—the TilePane. With the TilePane, rather than specifying the pane's width, you can use the more elegant approach and specify a value for the pane's number of columns. You can do that in a style sheet with a -fx-pref-columns: 3; property-value pair, or you can call pane.setPrefColumns(3);.

With both the FlowPane and the TilePane, if the user resizes the pane's window, the buttons will move to accommodate the window's new size. For example, if the user makes the window wider, the fourth button will move up to the right side of the first row. That would ruin the game! You can prevent such resizing by including the statement stage.setResizable(false); But there's another problem. With the original-size window, the window will be too small to fully display the Tic-Tac-Toe title at the top. Here's what we're talking about with a TilePane implementation:



©JavaFX

So what's the solution? If you widen the window by applying the -fx-pref-width property to the pane in a style sheet, that would affect the window the same as if the user resized the window—the fourth button would move to the first row when the window becomes wide enough for it to fit there. As an alternative, you could embed your FlowPane or TilePane in a surrounding pane large enough to fully display the Tic-Tac-Toe title. You'll learn how to embed a pane in another pane in the next section. For now, let's go with a different type of pane—the GridPane. Going back to Figure 18.15, you can see the result when a GridPane is used. We're able to specify a wider pane without affecting the number of buttons in a row. With the wider pane, the window's full title, Tic-Tac-Toe, displays.

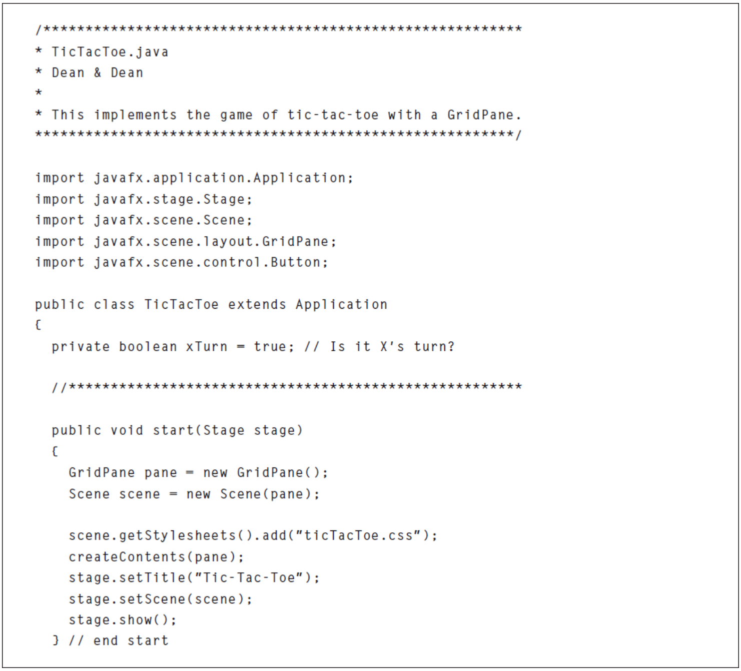
Figures 18.16a and 18.16b show the GridPane TicTacToe program. Figure

18.16a contains imports, an instance variable, and the start method. The imports should be familiar to you by now. The instance variable, xTurn, is a Boolean, and it keeps track of whose turn it is. The start method does the usual stuff that you've seen many times, but note that it instantiates a GridPane and loads the ticTacToe.css style sheet.

Figure 18.16b shows the TicTacToe program's createContents helper method. Note how it uses nested for loops to iterate through the board's three rows and three columns. Inside the for loops, note how we instantiate a button and then register an event handler for that button.

Let's slow down and examine the button's event handler in depth. We implement the event handler with a lambda expression. Note how we declare a new variable btn within the lambda expression to store the retrieved clicked button. You might think the lambda expression could simply use button, which was declared as a local variable in the createContents method, and not bother to declare a new btn variable. But alas, if we try to do that, the compiler complains, saying, “local variables referenced from a lambda expression must be final or effectively final.” This is the same issue we had in the DanceRecital program’s lambda expression. The problem is that it’s illegal to update a variable that’s declared as a local variable in the method that surrounds the lambda expression, and we updated the button by calling setText for it. The solution is to use an instance variable for btn or declare it locally within the lambda expression. Because there’s no need to use it outside of the lambda expression, we declare it in the lambda expression, with this statement:

```
Button btn = (Button) e.getSource();
```



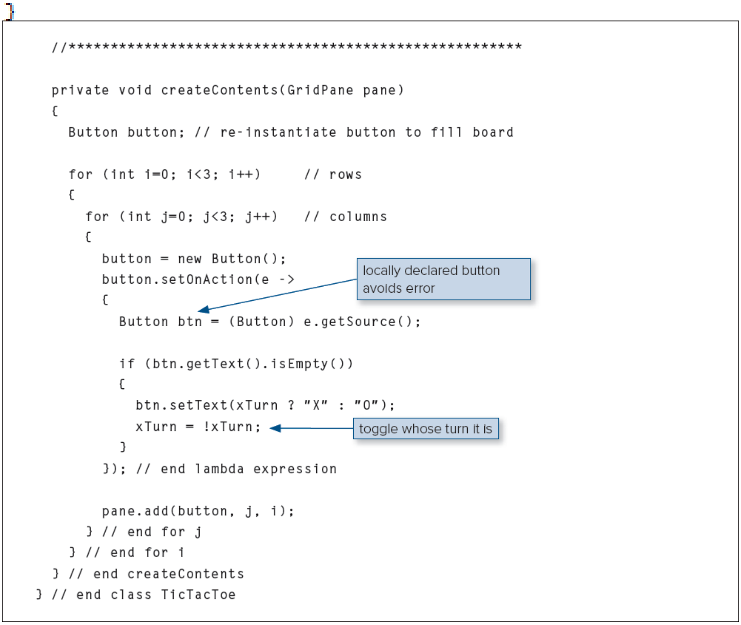
```
/***********************************************************
* TicTacToe.java
* Dean & Dean
*
* This implements the game of tic-tac-toe with a GridPane.
***********************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.GridPane;
import javafx.scene.control.Button;

public class TicTacToe extends Application
{
  private boolean xTurn = true; // Is it X's turn?

  //*********************************************************

  public void start(Stage stage)
  {
    GridPane pane = new GridPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add("ticTacToe.css");
    createContents(pane);
    stage.setTitle("Tic-Tac-Toe");
    stage.setScene(scene);
    stage.show();
  } // end start
```

Figure 18.16a TicTacToe program—part A

We use the (Button) cast operator because if there were no cast operator, the compiler would generate an error. Why? Because the compiler would see an Object at the right being assigned into a Button at the left. It sees an Object at the right because getSource has an Object return type. In this case, since getSource really returns a Button, it's legal to cast its returned value to Button, and that satisfies the compiler and eliminates the error.

In the lambda expression, look at this code fragment:

```
if (btn.getText().isEmpty())
{
  btn.setText(xTurn ? "X" : "O");
  xTurn = !xTurn;
}
```



```
  //*******************************************************

  private void createContents(GridPane pane)
  {
    Button button; // re-instantiate button to fill board

    for (int i=0; i<3; i++)     // rows
    {
      for (int j=0; j<3; j++)   // columns
      {
        button = new Button();
        button.setOnAction(e ->
        {
          Button btn = (Button) e.getSource();

          if (btn.getText().isEmpty())
          {
            btn.setText(xTurn ? "X" : "O");
            xTurn = !xTurn;
          }
        }); // end lambda expression

        pane.add(button, j, i);
      } // end for j
    } // end for i
  } // end createContents
} // end class TicTacToe
```

locally declared button avoids error

toggle whose turn it is

Figure 18.16b TicTacToe program—part B

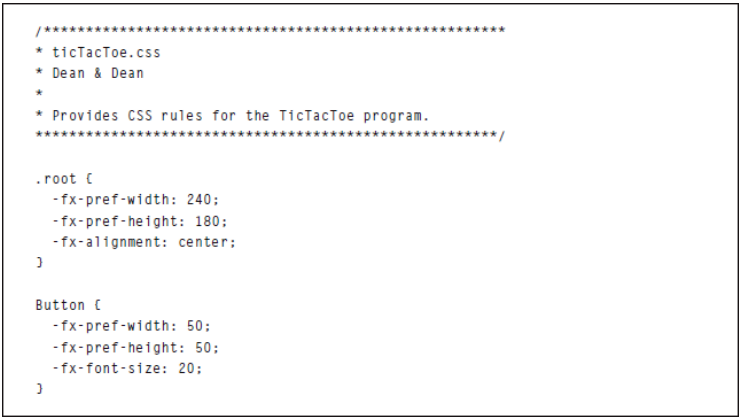
The if condition checks for a blank button, and if it's blank, we assign the appropriate letter (X or O) to the button's label and then toggle the Boolean variable, xTurn. The xTurn variable is an instance variable, so it's legal to update its value (by calling setText) inside the lambda expression.

Below the lambda expression is the code that adds the button to the pane:

```
pane.add(button, j, i);
```

j before i – what's up with that? You might recall from our earlier discussion of the GridPane container that the constructor call's second and third arguments specify the column and row, respectively, where the component goes. If you look at the program's nested for loops, you'll see that j holds the column value and i holds the row value, so all is well.

Now take a look at the program's style sheet, shown in Figure 18.17. Note how the .root rule determines the window’s initial dimensions and the GridPane’s center alignment. Note how the Button rule (no dot in front of Button, which indicates a type selector rule) determines the buttons’ dimensions and label font size.



```
/********************************************************
* ticTacToe.css
* Dean & Dean
*
* Provides CSS rules for the TicTacToe program.
********************************************************/

.root {
  -fx-pref-width: 240;
  -fx-pref-height: 180;
  -fx-alignment: center;
}

Button {
  -fx-pref-width: 50;
  -fx-pref-height: 50;
  -fx-font-size: 20;
}
```

Figure 18.17 TicTacToe program’s style sheet

18.8 Embedded Panes, HBox, and Math Calculator Program

Suppose you’d like to implement a GUI

program that prompts the user to enter a number and calculates the square root or the logarithm of that number depending on what button the user clicks. More specifically, generate the window shown in Figure 18.18. The user enters a value for x in the left text box and clicks the sqrt x or log10 x button to generate the square root of x or the base-10 logarithm of x, respectively. What type of layout scheme should you use? Coming up with a good layout scheme often requires some thought, so let's take some time to walk through the creative process for this program.

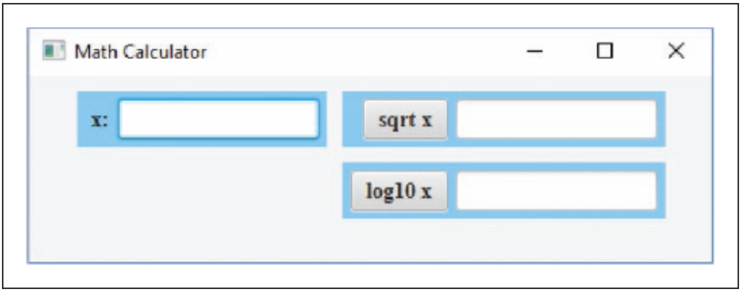


Figure 18.18 Window generated by the MathCalculator program

©JavaFX

From Alternate Layouts to Embedded Layouts

In Figure 18.18, note how the math calculator window's logarithm components (button and text box) are directly below the square root components. They are horizontally aligned. That suggests that a pane with a row-column orientation is the way to go. That pretty much eliminates the FlowPane, VBox, BorderPane, and TextFlow panes.

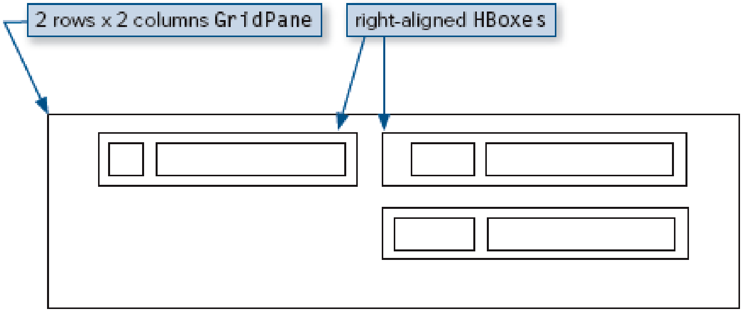
The math calculator window appears to have two rows and four columns. So, is a two-row by four-column TilePane scheme appropriate? Remember that with a TilePane, components are added left-to-right top-to-bottom in sequential order. So to position the logarithm components in the lower right tiles, you would need to first add invisible components (empty labels would work) to the bottom-left tiles. With the TilePane's left-to-right top-to-bottom component positioning strategy, that means if the user widens the window sufficiently, then the second row's first component would move up and be positioned at the end of the first row. That would lead to a seriously ugly layout.

The TilePane generates same-sized cells. That would be okay for most of the cells in the math calculator window, but not for the top-left cell. The top-left cell holds the x: label. With such a small label, we want a relatively small cell for it, but that's not an option for a TilePane.

With all these problems for the TilePane, it's time to move on. With a row-column structure, the GridPane might work. Remember that with a GridPane, you specify the location where you want to add a component, so there would be no need to add dummy invisible components to the lower left corner before adding the logarithm components to the lower right corner. Unlike a TilePane, GridPane columns can have different widths. Column widths conform to the widths of the components in the column. So for the top-left cell, which holds the narrow x: label, the column would be narrow. Yay!

Glance back at Figure 18.18 and notice that there are three pairs of components—each one with a label at its left and then either a text box or a button. To achieve a visual grouping for each pair, notice how there's a merged background color for each pair, with no background color between the pairs. With a two-row by four-column GridPane, that background color scheme would be difficult to achieve. The solution? Use a two-row by two-column GridPane with three subordinate panes, one for each of the three pairs of components.

A *wireframe* is a skeletal framework that shows where components go within a window. For a nontrivial window, it's sometimes helpful to first draw a wireframe and use it to help you decide what layout panes to use for the window. Here's the wireframe for the math calculator program's window, with callouts that show what we have in mind for the layout panes:



So what type of pane should we use for the GridPane's three subordinate panes? We could use a FlowPane, but that would leave the subordinate panes' contents vulnerable to wrap-around. A better choice is an HBox. An HBox is a horizontal version of the

VBox you saw earlier. Components are still added sequentially, as in a FlowPane or VBox, but with an HBox everything stays on the same row. So, what we end up with is three HBoxes in a GridPane. Voila, panes of one type inside a pane of another type. Pretty cool, eh?

When you have a nontrivial window, it’s very common to have embedded panes. When that happens, it might take some tweaking to get your windows to look right. Despite the tweaking, using embedded panes is a lot easier than having to manually position many individual components with pixel values.[5]

MathCalculator Program

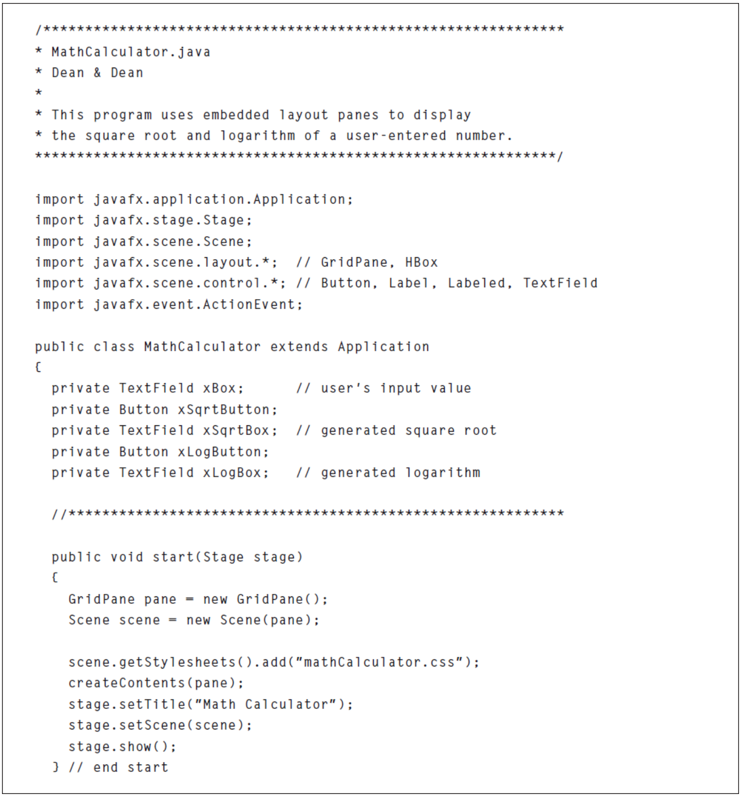
Now look at the MathCalculator program listing in Figures 18.19a–e. Figure 18.19a shows imports, instance variables, and the start method. The start method instantiates a GridPane, loads the program's style sheet, and calls the createContents helper method. Pretty standard stuff.

Figure 18.19b shows the createContents method. Note how it declares three HBox pane variables—xPane for the user input area, xSqrtPane for the square root area, and xLogPane for the logarithm area. For each pane variable, it calls the helper method getSubPane, which is in charge of creating and populating the subordinate pane. After generating the three subordinate panes, the createContents method adds them to the surrounding GridPane. For example, here's how the first subordinate pane gets added to the GridPane's top-left cell:

```
pane.add(xPane, 0, 0);
```

Finally, the createContents method registers an event handler for the square root button and the logarithm button. In both cases, the event handler uses a method reference that refers to the handle method.

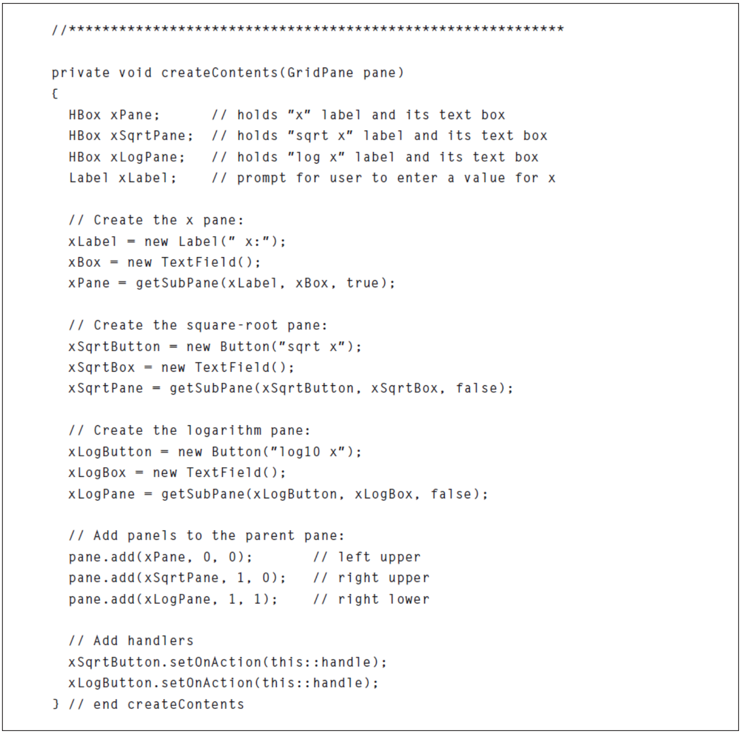
Take a look at the getSubPane helper method in Figure 18.19c. It receives three parameters—two components (a prompt label or a button, and then a text box) and then a Boolean variable that determines whether the text box should be editable. Only the first subordinate pane is editable, so only the first call to getSubPane specifies true for its third argument. The method first instantiates an HBox pane, then adds its two passed-in components to the pane. It then takes care of a couple formatting issues. If the user resizes the window to make it narrower, it's OK for the text boxes to narrow because the user can still enter input into the text boxes unencumbered. But we don't want the prompt label or the buttons to narrow because that might obfuscate the user interface. Therefore, we call labeled.setMinWidth(Labeled.USE_PREF_SIZE to prevent the prompt label or button from becoming less wide than its natural width. The second formatting issue is a call to setEditable to make the text box editable or read-only, depending on the value of the passed-in third parameter. Normally, we prefer using a style sheet for formatting, but style sheet rules would not be able to handle these two formatting issues in an elegant manner. All the other formatting issues are addressed in the program's style sheet, and we'll examine the style sheet's CSS rules after completing our walk-through of the program's java code.



```
/****************************************************************
* MathCalculator.java
* Dean & Dean
*
* This program uses embedded layout panes to display
* the square root and logarithm of a user-entered number.
****************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;  // GridPane, HBox
import javafx.scene.control.*; // Button, Label, Labeled, TextField
import javafx.event.ActionEvent;

public class MathCalculator extends Application
{
  private TextField xBox;      // user's input value
  private Button xSqrtButton;
  private TextField xSqrtBox;  // generated square root
  private Button xLogButton;
  private TextField xLogBox;   // generated logarithm

  //*************************************************************

  public void start(Stage stage)
  {
    GridPane pane = new GridPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add(”mathCalculator.css”);
    createContents(pane);
    stage.setTitle(”Math Calculator”);
    stage.setScene(scene);
    stage.show();
  } // end start
```

Figure 18.19a MathCalculator program—part A

Figure 18.19d shows the handle method, which serves as the event handler for the two buttons. Note the try and catch blocks, which attempt to read the user's input and store it in the double variable, x. If the user enters a value that's not a double, then −1 is assigned to x, with the −1 serving as a flag for invalid input. After using getSource to determine which button was clicked, the handle method checks for a negative user input (which is invalid) and displays an error message accordingly. If the user input is -nonnegative, the method calls either Math.sqrt(x) or Math.log10(x) (depending on which button was clicked) and puts the result in the appropriate read-only text box. For example, here's how we put the result in the square root text box:

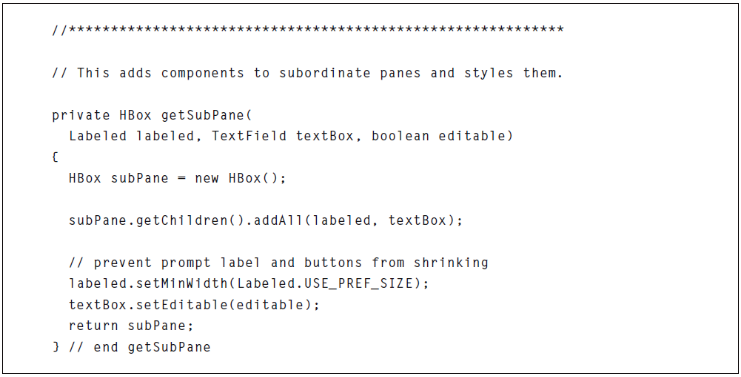


```
//************************************************************

private void createContents(GridPane pane)
{
  HBox xPane;      // holds ”x” label and its text box
  HBox xSqrtPane;  // holds ”sqrt x” label and its text box
  HBox xLogPane;   // holds ”log x” label and its text box
  Label xLabel;    // prompt for user to enter a value for x

  // Create the x pane:
  xLabel = new Label(” x:”);
  xBox = new TextField();
  xPane = getSubPane(xLabel, xBox, true);

  // Create the square-root pane:
  xSqrtButton = new Button(”sqrt x”);
  xSqrtBox = new TextField();
  xSqrtPane = getSubPane(xSqrtButton, xSqrtBox, false);

  // Create the logarithm pane:
  xLogButton = new Button(”log10 x”);
  xLogBox = new TextField();
  xLogPane = getSubPane(xLogButton, xLogBox, false);

  // Add panels to the parent pane:
  pane.add(xPane, 0, 0);        // left upper
  pane.add(xSqrtPane, 1, 0);    // right upper
  pane.add(xLogPane, 1, 1);     // right lower

  // Add handlers
  xSqrtButton.setOnAction(this::handle);
  xLogButton.setOnAction(this::handle);
} // end createContents
```

Figure 18.19b MathCalculator program—part B

xSqrtBox.setText(String.format("%7.5f", result));

See the String.format method call? As always, you want your output to look good, and the format method does just that. It works the same as the printf method except that instead of printing a formatted value, it returns a formatted value. In the handle method, we call String.format to retrieve a formatted version of the calculated logarithm value. Specifically, the %7.5f conversion specifier returns a floating-point value with five decimal places and seven total characters.



```
//*************************************************************

// This adds components to subordinate panes and styles them.

private HBox getSubPane(
  Labeled labeled, TextField textBox, boolean editable)
{
  HBox subPane = new HBox();

  subPane.getChildren().addAll(labeled, textBox);

  // prevent prompt label and buttons from shrinking
  labeled.setMinWidth(Labeled.USE_PREF_SIZE);
  textBox.setEditable(editable);
  return subPane;
} // end getSubPane
```

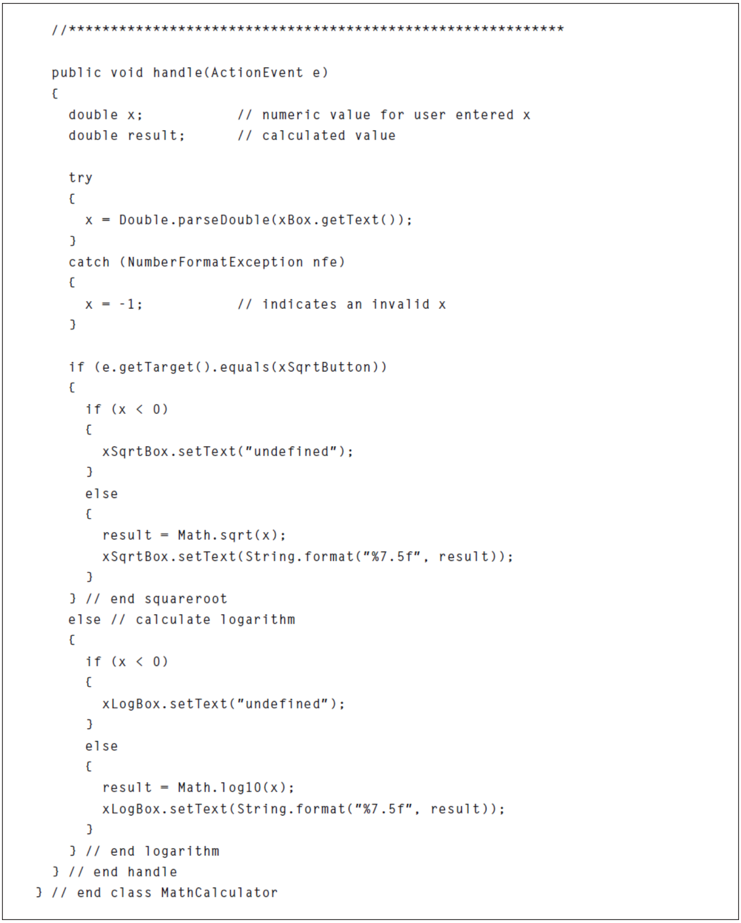
Figure 18.19c MathCalculator program—part C

Let's now focus on the MathCalculator program's format. Glance back at the program's window in Figure 18.18 and note the layout for the four grid areas for the two-row by two-column GridPane. Specifically, note the padding for the perimeter's interior, the gaps between the four grid areas, and the top-center alignment for the grid areas. Those attributes are not provided by default; for the GridPane, the default is no padding, no gaps, and top-left alignment. For the MatchCalculator program, we provide those attributes in the program's style sheet, shown in Figure 18.19e. Specifically, in the .root rule, the -fx-padding, -fx-hgap, -fx-vgap, and -fx-alignment properties take care of padding, horizontal and vertical gaps, and alignment.

Turning to the HBox subordinate panes, glance again at the program's window and note the layout for the components within each of the three subordinate panes. Each subordinate pane has padding, horizontal gap, and right alignment. As with the GridPane, the HBox has defaults of no padding, no gaps, and top-left alignment. We override those defaults in the program's style sheet. Specifically, in the HBox type selector rule, the -fx-padding, -fx-spacing, and -fx-alignment properties take care of padding, the horizontal gap, and alignment. There's also an -fx-background-color property, which assigns light sky blue to the subordinate pane's background color.

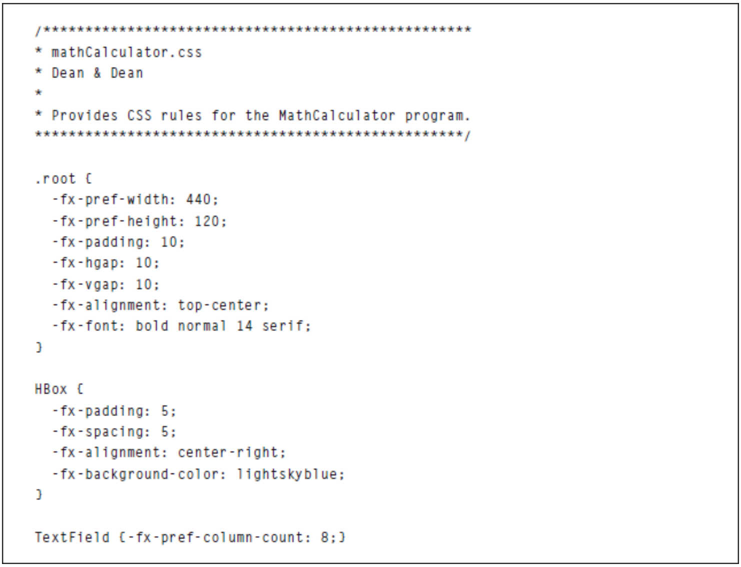
In instantiating its HBox containers, the MathCalculator program calls the 0-parameter HBox constructor. As an alternative, you can call the HBox constructor's 1-parameter constructor and pass in a value that represents the spacing in pixels between the HBox container's components. For example:

```
HBox subPane = new HBox(5);
```



```
  //***********************************************************

  public void handle(ActionEvent e)
  {
    double x;           // numeric value for user entered x
    double result;      // calculated value

    try
    {
      x = Double.parseDouble(xBox.getText());
    }
    catch (NumberFormatException nfe)
    {
      x = -1;           // indicates an invalid x
    }

    if (e.getTarget().equals(xSqrtButton))
    {
      if (x < 0)
      {
        xSqrtBox.setText(”undefined”);
      }
      else
      {
        result = Math.sqrt(x);
        xSqrtBox.setText(String.format(”%7.5f”, result));
      }
    } // end squareroot
    else // calculate logarithm
    {
      if (x < 0)
      {
        xLogBox.setText(”undefined”);
      }
      else
      {
        result = Math.log10(x);
        xLogBox.setText(String.format(”%7.5f”, result));
      }
    } // end logarithm
  } // end handle
} // end class MathCalculator
```

Figure 18.19d MathCalculator program—part D



```
/*****************************************************
* mathCalculator.css
* Dean & Dean
*
* Provides CSS rules for the MathCalculator program.
*****************************************************/

.root {
  -fx-pref-width: 440;
  -fx-pref-height: 120;
  -fx-padding: 10;
  -fx-hgap: 10;
  -fx-vgap: 10;
  -fx-alignment: top-center;
  -fx-font: bold normal 14 serif;
}

HBox {
  -fx-padding: 5;
  -fx-spacing: 5;
  -fx-alignment: center-right;
  -fx-background-color: lightskyblue;
}

TextField {-fx-pref-column-count: 8;}
```

Figure 18.19e MathCalculator program's style sheet

18.9 Plain Pane Container and Component Positioning

Pane Class

Figure 18.1 shows that under Region at the top of the container hierarchy is the Pane class. In our examples so far, we've used descendants of the Pane class (FlowPane, GridPane, etc.), but not the Pane class itself. The Pane class can store multiple components, but in doing so, it doesn't attempt to position them in a pleasing manner. That can be a good thing if you want exclusive control over where the container's components go. All you need to remember is that the default component position is in Pane's upper left corner, and to the extent that components overlap, subsequently added components cover previously added components.

Since Pane is a Region descendant, its objects can use CSS properties like -fx-background-color. And since Pane is a Node descendant, its objects can still use CSS properties like -fx-translate-x, -fx-scale-x, and -fx-rotate.

A Pane can contain a Node object or any of its descendants. The Node class is the parent of the Parent class, and one of Parent’s siblings under Node is the Shape class. Since Shape is not a subclass of Parent, a Shape cannot have children, but Shapes like Circle and Rectangle are quite useful, nevertheless. Some Circle and Rectangle constructors include offset as well as size parameters, so when you add one of these shapes to a group, you can set its location and dimensions at the same time.

By default, most components are opaque. So, when you add another child to a pane, the new child covers all previously added children that happen to be under it. However, you can use one or more of the translation methods described below to move different components to different places, so that no two components share the same area. Then, the order of entry into the group doesn’t matter, and all components can be seen in their entirety. Sometimes you’ll want to position components on top of other ones to create more sophisticated structures. Then you can specify the amount of opaqueness in a continuous range from 0.0 (completely transparent) to 1.0 (completely opaque.) You can also specify color gradients. By carefully controlling relative locations, opaqueness, and color gradients, you can create some very interesting effects.

Pane Component Positioning

As indicated in Figure 18.1, Pane is the common parent of many different types of containers. By default, a Pane container positions its Node components on top of each other in its upper left corner. For rectangular components, that means the component's upper left corner coincides with the pane's upper left corner. For an ellipse or circle, that means the component's center coincides with the pane's upper left corner. Usually, such component positioning is unsatisfactory, so with a plain Pane container, we usually employ explicit program code to specify a desired position for each component.

The Node class defines two types of component positioning, layout positioning and translate positioning. Layout positioning establishes relatively stable reference positions. Translate positioning creates dynamic deviations from layout positions. At any time, the current position is the layout position plus the translate position.

Each Node object remembers its position with its layoutX, layoutY, translateX, and translateY properties. The node's layoutX and layoutY properties are determined by the layout pane in which it resides. The layout pane leave its nodes' translateX and translateY values at zero. And for the Pane container, it leaves all four of its nodes' positioning properties at zero. Subsequent program code may alter default node positioning by having particular nodes assign nonzero values to their translateX and translateY properties.

A program that uses a Pane container should explicitly position its Pane container components by having each component call setTranslateX(double x) and setTranslateY(double y), where x and y are the desired horizontal and vertical displacements.

When there's user interaction, a component's current position may be unknown. Then, when a desired new position depends on the current position, a component will need to determine its actual current position. It can do that by calling one or more of its methods, getLayoutX(), getLayoutY(), getTranslateX(), and getTranslateY().

By default, a Pane's width and height conform to the width and height of the Pane's container, which might be, for example, another Pane or a Scene. If sizing or positioning of an embedded Pane's components make portions of some components extend beyond the boundaries of the embedded Pane, those portions will also extend beyond the boundaries of the embedded Pane's container, and the embedded pane's container will clip off those extensions. What you'll see will be only those component portions that fall within the range of the embedded pane's container. This feature enables a program to portray components as entering or leaving a current field of view. Cool!

Summary

- Layout panes automate the positioning of components.

- The FlowPane class implements a simple one-compartment layout scheme and allows more than one component to be inserted into the compartment.
- The GridPane class implements a rectangular grid, and the programmer assigns each component to a particular column and row. A column's width comes from the width of the column's widest component. A row's height comes from the height of the row's tallest component.
- The HBox and VBox layouts align entries sequentially in one row or one column, respectively.
- The BorderPane layout provides five areas in which to insert

components—center, top, right, bottom, and left.

- The TilePane lays out its components in a rectangular grid of equal-sized cells. Each cell can hold only one component.
- If you have a complicated window with many components, you might want to compartmentalize them by embedding containers in other containers.
- A Pane is a stack of Nodes whose relative positions are not set by the Pane container; instead, they’re set by explicit program code.
- Whereas a Group contracts to snugly fit its contents, a Pane expands to fill its surrounding container.

Review Questions

§18.2 Layout Panes

1. JavaFX layout panes are designed to adjust their components' positions if the user resizes the layout pane's window or if the code resizes the layout pane's components. (T / F)

2. Which package holds the layout pane classes?

§18.3 FlowPane and GridPane—Competing Layout Philosophies

3. How does the FlowPane arrange its components?

4. Assume you've instantiated a FlowPane or GridPane container and assigned it to a variable named pane. Provide a single statement that causes pane to use center-right alignment for its components.

5. For the GridPane version of the Greeting program, write a statement that makes the greeting start under the name box, and allocate just enough cells to avoid resizing the text box if the name is long.

§18.4 VBox Program with Two Stages and an Image File

6. Provide a single statement that declares a stage named stage2 and initializes it to a new utility stage.

§18.5 BorderPane

7. What are the five areas in a BorderPane layout in the order used by the five-parameter constructor, and which areas get the corners?

8. The sizes of the five areas in a BorderPane are determined at runtime based on the contents of the four outer areas. (T / F)

9. By default, how many components can you put in any one area of a BorderPane?

10. Given the reference, BorderPane pane = new BorderPane(); write a single statement that adds a new Label with the text “Stop” to the center area of a BorderPane. The label should be centered within the center area.

§18.6 TilePane and TextFlow Containers

11. When you instantiate a TilePane, you should always specify both the number of rows and the number of columns. (T / F)

12. In a TilePane, all cells are the same size. (T / F)

§18.7 TicTacToe Program

13. What happens to the xTurn variable in the TicTacToe program if you click the same cell twice?

§18.8 Embedded Panes—HBox and MathCalculator Example

14. Why are embedded panes particularly useful in BorderPane, TilePane, and GridPane outer containers (as opposed to FlowPane, VBox, and HBox outer containers)?

§18.9 Plain Pane Container and Component Positioning

15. If you add a Circle to a plain Pane without specifying the Circle's position, where does the Circle go and how much of it is visible?

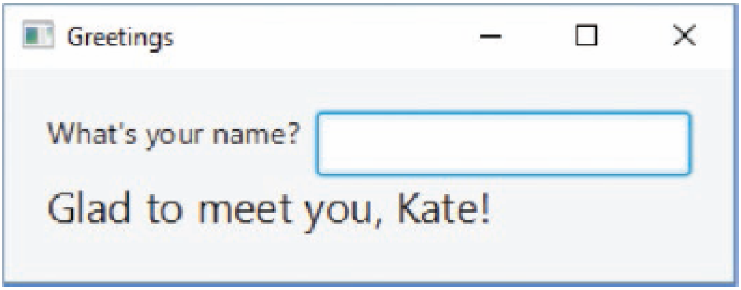
Exercises

1. [after §18.3] In a FlowPane, a button control expands so that it completely fills the size of the area in which it is placed. (T / F)

2. [after §18.3] Assume you have assigned a GridPane container object to a pane reference variable. Write a code fragment that fixes the width of the container's second column at 200 pixels but lets the GridPane establish the widths of all other columns.

3. [after §18.4] Rewrite the Recital program so that instead of using a secondary stage to obtain string values for the data array, it uses System.out.println to display each prompt in the console window and uses Scanner's nextLine method to retrieve the corresponding user input.

4. [after §18.5] Provide a complete program that is a modification of the lambda-expression version of Chapter 17's Greeting program. Your new program should use a BorderPane (instead of a FlowPane), and it should generate the following display after a name has been entered. Use defaults to make your code and the accompanying CSS file as simple as possible.



Glad to meet you, Kate!

©JavaFX

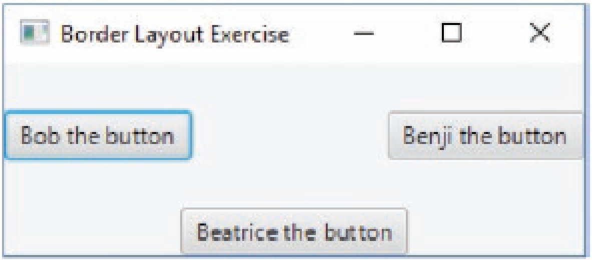
5. [after §18.5] In a BorderPane, what happens if the right area is empty? Said another way, which area(s), if any, expand(s) if the right area is empty?

6. [after §18.5] In the AfricanCountries program in Figure 18.10, suppose there is no setMaxSize method call. What would the resulting window look like?

7. [after §18.5] Assume you have this program:

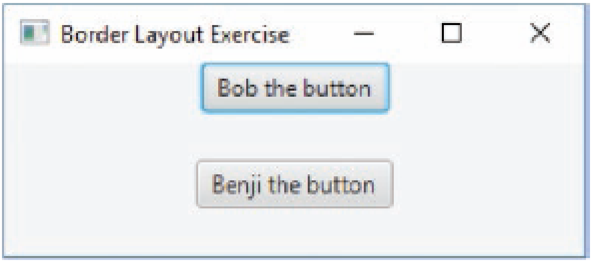
```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.control.Label;

public class BorderPaneExercise extends Application
{
  public void start(Stage stage)
  {
    BorderPane pane = new BorderPane();

    stage.setTitle("Border Layout Exercise");
    stage.setScene(new Scene(pane, 300, 100));
    pane.setTop(new Label("Lisa the label"));
    pane.setCenter(new Label("LaToya the label"));
    pane.setBottom(new Label("Lemmy the label"));
    stage.show();
  } // end start
} // end BorderPaneExercise
```

a) Modify the above code to produce this output:


Border Layout Exercise
Bob the button
Benji the button
Beatrice the button

©JavaFX

b) Modify the above code to produce this output:


Border Layout Exercise
Bob the button
Benji the button

©JavaFX

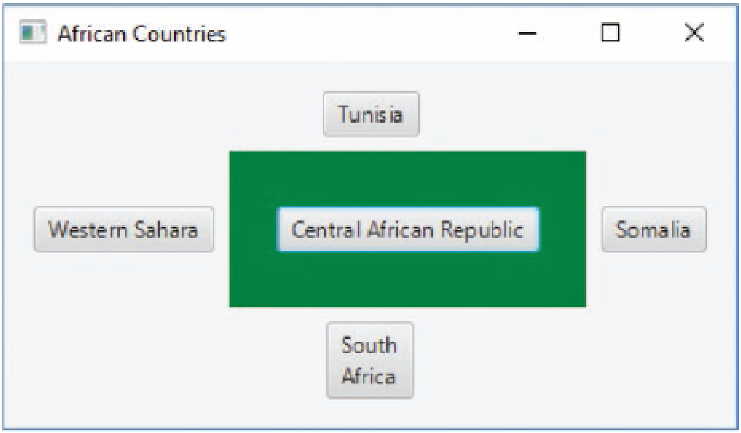
For both (a) and (b), call setAlignment(*component*, Pos.CENTER).

8. [after §18.6] If a Button component is directly added to a TilePane cell, it expands so that it completely fills the size of its cell. (T / F)

9. [after §18.6] Assume a TilePane is constructed with no specification of orientation. Given the following code fragment, draw a picture that illustrates the buttons’ positions within the program’s window.

```
public void start(Stage stage)
{
  TilePane pane = new TilePane();

  stage.setScene(new Scene(pane));
  pane.setPrefColumns(3);
  // This centers desired display in a minimum window
  pane.setStyle(”-fx-padding: 20; -fx-font-size: 16”);
  for (int i=0; i<7; i++)
  {
  pane.getChildren().add(new Button(Integer.toString(i+1)));
  } //end for i
  stage.show();
} // end start
```

10. [after §18.8] Modify the AfricanCountries program in Figure 18.10 in Section 18.5 to produce an initial display that looks like this:


African Countries
Tunisia
Western Sahara
Central African Republic
Somalia
South Africa

©JavaFX

Create the scene with a width of 400 and a height of 200. Embed a StackPane in each BorderPane compartment, and put the button in those embedded StackPanes. This automatically centers the buttons in their respective compartments. Specify all other styling in a CSS style sheet. On this style sheet, supply three style classes—root, child, and center. Register the center style class (which specifies the center compartment's green background) with the center compartment's StackPane, using a statement like this:

pane.getCenter().getStyleClass().add("ce

11. [after §18.8] Write a program that generates the CSS color names window shown in Figure 17.18. In your program, implement a TilePane container named table with setPrefColumns(4), setHgap(5), setVgap(2), and setStyle("-fx-padding: 5, 0, 0, 5"). For each tile in table, embed a FlowPane container named panel with setPrefWidth(183) and setStyle("-fx-font-size: 10;"). Each panel should hold two components: (1) a four-space Label named swatch that displays a color using a CSS color name, and (2) an anonymous Label that displays a description for the color swatch at its left. Set the swatch colors with setStyle("-fx-background-color: " + names[i]), where names is an array shown in the following program skeleton.

```
public class ColorTable extends Application
{
  private final int TABLE_WIDTH = 768;
  private final int TABLE_HEIGHT = 640;
  private String[] names = {
    <include color names here, from https://docs.oracle.com/javafx/2/api/javafx/scene/
    doc-files/cssref.html>
  };
  private String[] numbers = {
    <include color numbers here, from https://docs.oracle.com/javafx/2/api/javafx/scene/
    doc-files/cssref.html>
  };
  <start method goes here>
  <createContents method goes here>
} // end class ColorTable
```

12. [after §18.9] Write a program that uses setTranslateX and setTranslateY to position a bright blue circle of radius 50 in the center of a plain Pane that’s in a Scene of width 200 and height 150.

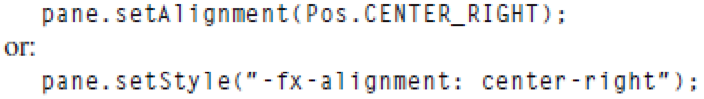
Review Question Solutions

1. True.

2. The javafx.scene.layout package holds the layout pane classes.

3. The FlowPane places components left-to-right in a row until it runs out of space, and then it goes to the next row and does the same thing, and so on.

4. Use either:



```
pane.setAlignment(Pos.CENTER_RIGHT);
```

or:

```
pane.setStyle("-fx-alignment: center-right");
```

5. pane.add(greeting, 1, 1, 2, 1);

6. Stage stage2 = new Stage(StageStyle.UTILITY);

7. The five areas of a BorderPane layout in the five-parameter constructor order are center, top, right, bottom, and left. The two top corners are in the top area, and the two bottom corners are in the bottom area.

8. True.

9. BorderPane, TilePane, and GridPane containers can have no more than one component per area.

10. pane.setCenter(new Label("Stop"));

(Center is the default alignment.)

11. False. If orientation is unspecified or horizontal, specify columns. If orientation is vertical, specify rows.

12. True. All cells of a TilePane are the same size.

13. Nothing. It does not change value.

14. Embedded panes are particularly useful in BorderPane, TilePane, and GridPane outer containers (as opposed to FlowPane, VBox, and HBox outer containers) because each BorderPane area or each TilePane or GridPane cell can contain only one component.

15. If you add a Circle to a plain Pane without specifying the Circle's position, the Circle's

> center goes at the upper left corner of the Pane. If this Circle is the final component added to the Pane, and if the Circle's radius does not exceed the width and height of the Pane's container, the lower right quadrant of the circle is visible. However, subsequently added opaque components may overlay and hide some or all of the visible part of the Circle.

---

[1] For lack of space, Figure 18.1 does not include three JavaFX classes used in the next chapter's LunarEclipse program—Stop (child of Object), CycleMethod (grandchild of Object under Enum), and BoxBlur (grandchild of Object under Effect). Also missing from Figure 18.1 is JavaFX's MouseEvent class, which is used in the next chapter's DragSmiley program.

[2] As explained in Chapter 17, the Image class is not a subclass of the Node

class, but ImageView is. So if you want to add an image to a pane, first wrap the Image object in an ImageView class by calling the ImageView constructor.

[3] The quotation is from an English translation of *The Brothers Karamazov* by Fyodor Dostoyevsky.

[4] The line wrap we've been describing works for the horizontal orientation found in Western languages. The TextFlow container does not support the vertical orientation found in traditional Chinese, Japanese, or Korean languages, where characters proceed down a column and wrap to the top of the next column.

[5] If you want to set component positions manually, perhaps because you don't want them to jump around, instead of embedding an instance of one of the descendants of the Pane class, you can embed an instance of the Group class or an instance of the Pane

class itself. Group (found in the javafx.scene package) and Pane (found in the javafx.scene.layout package) store components, like the other containers you've learned about, but they do not rearrange their components if something alters their window size. We provide details on the Pane container later in this chapter.

GUI Programming—Additional GUI Components, Additional Event Handlers, Animation

Objectives

- Learn the basics of user interface design.

- Implement a TextArea control for user text input that requires more than one line.
- Implement a CheckBox control for yes/no user input.
- Implement RadioButton and ComboBox controls when the user needs to choose a value from among a list of predefined values.
- Implement a menu system.
- Implement a scroll pane so the user can navigate through content that's larger than the window.
- Learn how to capture mouse events and write event handler code for them.
- Learn how to embed an image in a GUI window.
- Implement a slider so the user can manually translate a GUI component.

- Learn how to animate components in a GUI window.

Outline

19.1 Introduction 897

19.2 User Interface Design 897

19.3 TextArea Control 898

19.4 CheckBox and RadioButton Controls 903

19.5 ComboBox Control 909

19.6 Job Application Program 913

19.7 ScrollPane and Menu Classes 918

19.8 Images and Mouse Events 922

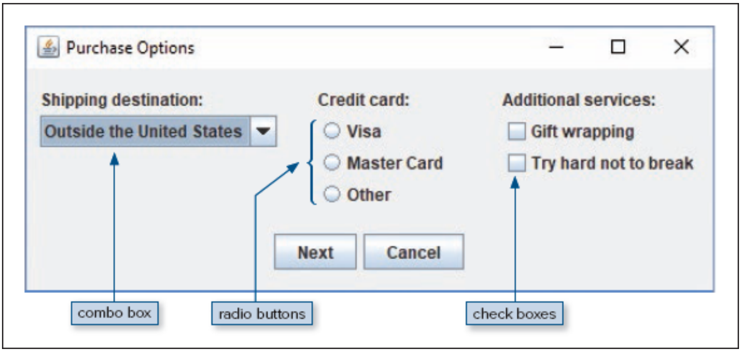
19.9 Lunar Eclipse Program with Circle, RadialGradient, and Slider 928

19.10 Animation 933

19.1 Introduction

This is the third chapter in our three-chapter sequence on GUI programming. In Chapter 17, we introduced you to a few GUI controls that provide basic input/output functionality—Label, TextField, and Button. In Chapter 18, we described different ways to organize those controls using different layout panes. In this chapter, we'll round out your GUI tool bag by introducing GUI controls that provide more advanced input/output functionality—TextArea, CheckBox, RadioButton, ComboBox, and Slider. In addition, you'll learn about scroll panes, menu systems, and animation.

For a preview of what some of the controls can do, see Figure 19.1. Note the combo box, radio button, and check box controls. Also note the grouping of radio buttons in the center, the grouping of check buttons on the right, and the grouping of the Next and Cancel buttons at the bottom center. In this chapter, you'll learn how to create those components and additional ones. Also, you'll get practice with the layout panes that you learned about in the last chapter by using them to group components and position the component groups appropriately within a window.



Purchase Options

Shipping destination: Outside the United States

Credit card:
○ Visa
○ Master Card
○ Other

Additional services:
☐ Gift wrapping
☐ Try hard not to break

Figure 19.1 Example window that uses a combo box, radio buttons, and check boxes

©JavaFX

19.2 User Interface Design

With text-based programs, it's relatively easy to tell users what to do. As a programmer, you just provide text instructions and the user enters input when prompted to do so. With GUI programs, telling users what to do is more subtle. As a programmer, you display a window with various components, set up event handlers, and then wait for the user to do something. It's important that your display is easy to understand; otherwise your users may be confused.

In general, *user interface design* (UID) is the process of making software that's easy to use and fun. Well, you might have trouble making an insurance claim application fun, but you get the idea. For a GUI program, try to follow these guidelines:

- Choose the right components.
- Position components appropriately.
- Be consistent.

A good programmer will anticipate users' needs and create an interface that meets those needs, by incorporating components (text, color, buttons, text boxes, pictures, etc.) that are intuitive. You want users to be able to quickly figure out how to use the window's content. To help in that regard, you should try to avoid clutter and focus on clear, concise words (and graphics, if appropriate) that describe the window's content. Choosing the right interactive controls (buttons, text boxes, check boxes, etc.) is important, but the controls' labels are also important. For each label, use just one word or a few words that get to the point quickly. Don't be afraid to remove unnecessary text and to have whitespace on your window. Whitespace can provide a nice respite for stressed-out tech-weary users. ☺

For your window's text, you should limit the number of fonts used. Pick pleasing fonts that go together well for your main content and your subsidiary content. Make sure that the foreground text colors contrast with the background colors so the text is easy to read. Normally, that means the contrasting colors should be different in terms of lightness and darkness. You should pick a set of colors for your text, background, and graphics that go together well.

In Figure 19.1's purchase form, we use radio buttons for the credit card selection and check boxes for the additional services selection. That's an example of choosing the right component. Radio buttons provide implicit instructions to the user about how to proceed. Most users recognize small circles as radio buttons, and know they should click one of them with their mouse. Likewise, most users recognize small gray squares as check boxes and know they can click one or more of them with their mouse.

Note the Next and Cancel buttons at the bottom center of the window. Assume that the window is one of several windows in a purchasing application. Assume that other windows in the application also display Next and Cancel buttons in the bottom center position. Placing Next and Cancel buttons in the same position is an example of being consistent. Consistency is important because users are more comfortable with things they've seen before. Be consistent with your fonts and color schemes. In a given application, if you choose red for a warning message, use red for all your warning messages.

In Figure 19.1, note how the three

radio button components (Visa, MasterCard, and Other) and the "Credit card:" label are positioned together as a group. More specifically, they're aligned in a vertical column and they're physically close together. That's an example of positioning components appropriately. Positioning them together as a group provides a visual cue that they're logically related. As another example of appropriate positioning, note that there are sizable gaps separating the left, center, and right component groups. Finally, note how the "Shipping destination:", "Credit card:", and "Additional services:" labels are aligned in the same row. That alignment, the aforementioned gaps, and the aforementioned component groupings all lead to a more appealing and understandable display.

19.3 TextArea Control

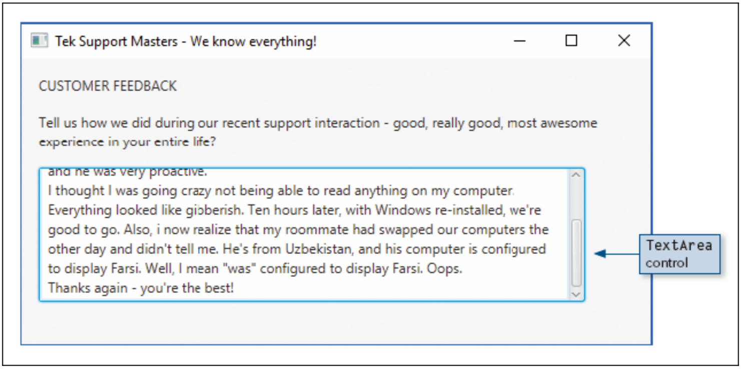
User Interface

For this chapter's first GUI component deep dive, we start with the TextArea control. Whereas the TextField control is for single-line text input, the TextArea control accommodates multiple-line text input. Note the example in Figure 19.2.

By default, TextArea controls are fairly large. In the program used to generate Figure 19.2, we reduced the TextArea's default size by specifying a column count of 40 and a row count of 6 (yes, we know, the column and row values aren't all that close to what actually displays). If the user types more than can fit on one line of the control, the default behavior is for the prior text to scroll off to the left, so it's unseen, and a horizontal scroll bar will be generated automatically. In Figure

19.2's TextArea control, there's no horizontal scroll bar because we call the TextArea class's setWrapText method with an argument value of true. But there is a vertical scroll bar. Vertical scroll bars are automatically generated when the user types more than can fit in all the rows in the TextArea.

Page 899



CUSTOMER FEEDBACK

Tell us how we did during our recent support interaction - good, really good, most awesome experience in your entire life?

and he was very proactive.
I thought I was going crazy not being able to read anything on my computer. Everything looked like gibberish. Ten hours later, with Windows re-installed, we're good to go. Also, i now realize that my roommate had swapped our computers the other day and didn't tell me. He's from Uzbekistan, and his computer is configured to display Farsi. Well, I mean "was" configured to display Farsi. Oops.
Thanks again - you're the best!

Figure 19.2 Customer feedback form that uses a TextArea control

©JavaFX

Implementation

To create a TextArea control, call the TextArea constructor like this:

TextArea *TextArea-reference* = new TextArea(*display-text*);

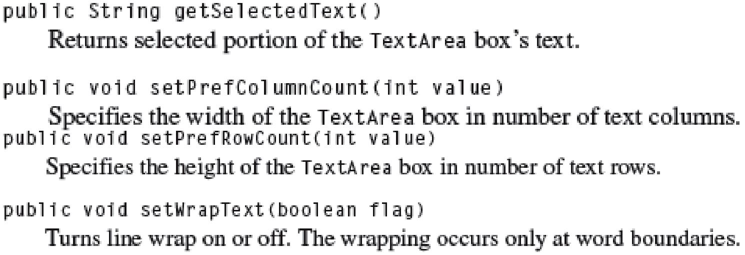
The *display-text* is the text that appears in the TextArea control initially. If the *display-text* argument is omitted, then the TextArea control displays no initial text.

Methods

The TextArea class is a subclass of the TextInputControl class. That's also the case for the TextField class. Thus, TextArea and TextField objects inherit the same methods. What follows are the inherited TextInputControl methods that you learned about in Chapter 17:

appendText, getLength, getText, setEditable, setPromptText, setText

In addition to those methods, here are some of the more popular methods defined in the TextArea class itself:



`public String getSelectedText()`

Returns selected portion of the `TextArea` box's text.

`public void setPrefColumnCount(int value)`

Specifies the width of the `TextArea` box in number of text columns.

`public void setPrefRowCount(int value)`

Specifies the height of the `TextArea` box in number of text rows.

`public void setWrapText(boolean flag)`

Turns line wrap on or off. The wrapping occurs only at word boundaries.

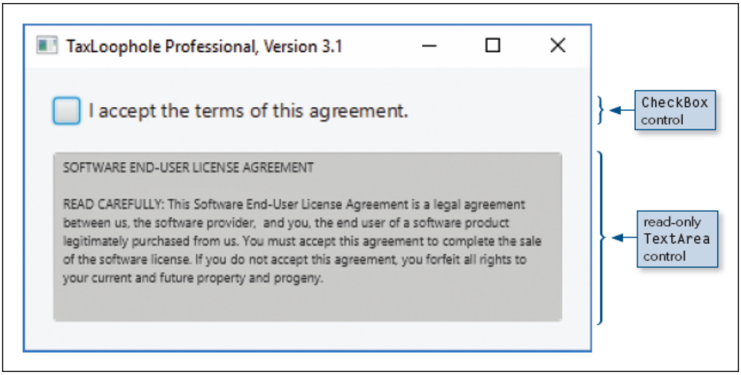
Page 9

Regardless of whether a TextArea control has been made to be read only (by calling setEditable(false)), the user can always use his/her mouse to select text within the TextArea box. There are two methods that deal with such selected text. The getSelectedText method, shown above, retrieves the selected text. The replaceSelection method, defined in the TextInputControl class, replaces the selected text with the string argument supplied by the replaceSelection method call. If there's no selected text, then the string argument gets inserted at the cursor position within the TextArea box. Here's the API heading for the replaceSelection method:

public void replaceSelection(String replacement)

Software License Agreement Program

See Figure 19.3. It shows one of those annoying disclaimer forms that you’re told to read before you install something. The small box in the top-left corner is a CheckBox control. We’ll describe CheckBox controls in the next section, but for now, let’s focus on the TextArea control at the bottom.


TaxLoophole Professional, Version 3.1
☐ I accept the terms of this agreement.
CheckBox control
read-only TextArea control

SOFTWARE END-USER LICENSE AGREEMENT

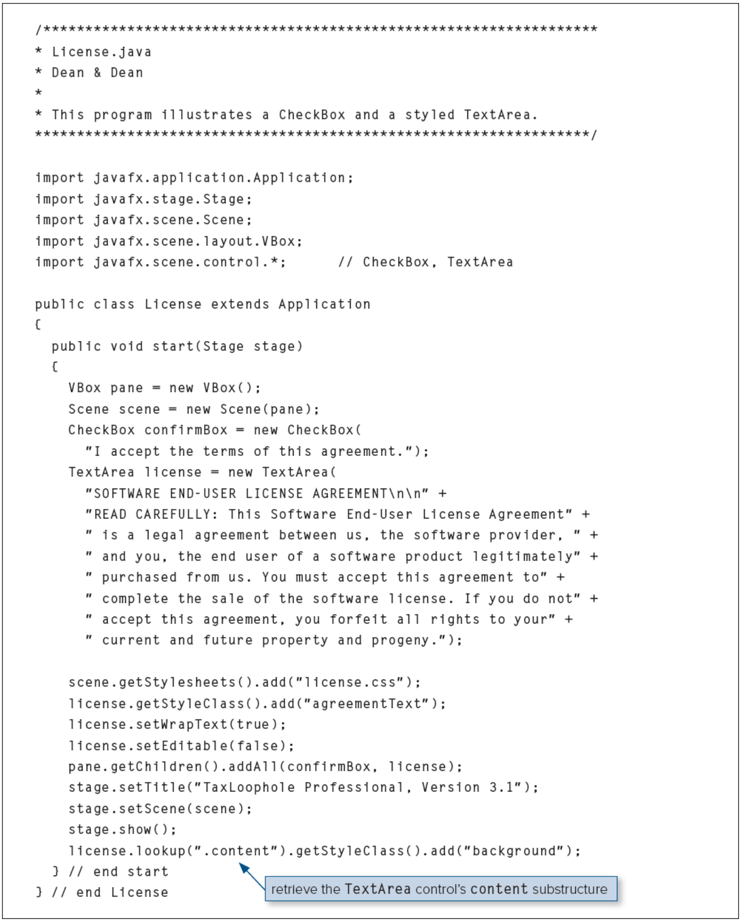
READ CAREFULLY: This Software End-User License Agreement is a legal agreement between us, the software provider, and you, the end user of a software product legitimately purchased from us. You must accept this agreement to complete the sale of the software license. If you do not accept this agreement, you forfeit all rights to your current and future property and progeny.

Figure 19.3 A window with a TextArea component and a CheckBox control

©JavaFX

We generated Figure 19.3’s window

with the License program, whose code is in Figure 19.4. You can see that we use a VBox pane to hold the two controls. The disclaimer text is for reading, not for user input, so we make the TextArea control read only by calling setEditable(false). In the resulting window, note that the TextArea's text is gray. We say that it's *grayed out*. Gray is the standard GUI way of telling the user that something is not editable. To mimic that behavior, we had to do a little extra work.



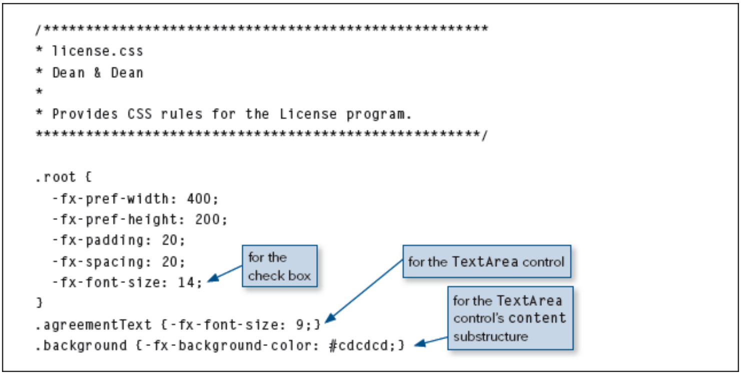
```
/*******************************************************************
* License.java
* Dean & Dean
*
* This program illustrates a CheckBox and a styled TextArea.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.VBox;
import javafx.scene.control.*;      // CheckBox, TextArea

public class License extends Application
{
  public void start(Stage stage)
  {
    VBox pane = new VBox();
    Scene scene = new Scene(pane);
    CheckBox confirmBox = new CheckBox(
      "I accept the terms of this agreement.");
    TextArea license = new TextArea(
      "SOFTWARE END-USER LICENSE AGREEMENT\n\n" +
      "READ CAREFULLY: This Software End-User License Agreement" +
      " is a legal agreement between us, the software provider, " +
      " and you, the end user of a software product legitimately" +
      " purchased from us. You must accept this agreement to" +
      " complete the sale of the software license. If you do not" +
      " accept this agreement, you forfeit all rights to your" +
      " current and future property and progeny.");

    scene.getStylesheets().add("license.css");
    license.getStyleClass().add("agreementText");
    license.setWrapText(true);
    license.setEditable(false);
    pane.getChildren().addAll(confirmBox, license);
    stage.setTitle("TaxLoophole Professional, Version 3.1");
    stage.setScene(scene);
    stage.show();
    license.lookup(".content").getStyleClass().add("background");
  } // end start
} // end License
```

retrieve the `TextArea` control's `content` substructure

Figure 19.4 License program with CheckBox and TextArea components in a VBox container

In the License program, verify that license is the name of the TextArea control. In the program's style sheet (Figure 19.5), verify that background is the name of the selector that applies a background color value of #cdcdcd, which is gray. Therefore, you might think that the following statement would connect the license control to the gray background selector:

```
license.getStyleClass().add("background");
```



```
/*******************************************************
* license.css
* Dean & Dean
*
* Provides CSS rules for the License program.
*******************************************************/

.root {
  -fx-pref-width: 400;
  -fx-pref-height: 200;
  -fx-padding: 20;
  -fx-spacing: 20;
  -fx-font-size: 14;
}
.agreementText {-fx-font-size: 9;}
.background {-fx-background-color: #cdcdcd;}
```

for the check box

for the TextArea control

for the TextArea control's content substructure

Figure 19.5 Style sheet for the License program

Nope. Sorry—some CSS rules need to be applied to a control's *substructure* rather than the control itself. If you look up TextArea in the JavaFX CSS API library, you'll see that the TextArea control has two substructures —scrollbar and content. The scrollbar substructure is, of course, for the control’s scrollbar(s). The content substructure is for the content inside the control. For our License program, we need to retrieve the content substructure first and then call getStyleClass and add as you’ve done in the past. Here’s how we do that, using the lookup method:

```
license.lookup(".content").getStyleClass().a
```

As you might surmise, the lookup method searches for the specified substructure (in this case, the content substructure) within the calling object and returns it if it’s found. Note where the lookup method call is within the License program – at the bottom. That’s because the JVM can access substructure objects (with the lookup method) only after the stage.show method call occurs.

In addition to wanting the TextArea’s text grayed out, we want it small. After all, the text is a license agreement disclaimer. You don't want the reader to actually read it, do you? In the License program's style sheet, note that the small text .agreementText rule is separate from the background color .background rule. That's because the -fx-font-size property must be applied to the TextArea control, not the content substructure. Thus, there's no lookup method call, just this run-of-the-mill standard code:

```
license.getStyleClass().add("agreementText
```

The style sheet's .root rule is pretty straightforward. The width, height, padding, and spacing properties all apply to the VBox pane. The font size property flows down to the check box control and applies to it. More specifically, it applies to the check box's label. It would normally also flow down to the TextArea control, but the TextArea control has its own

.agreementText rule that overrides the other rule.

19.4 CheckBox and RadioButton Controls

CheckBox Control

A *check box* control displays a small square with a label at its right. The label describes an option that the user can select by clicking on the check box. When the square contains a check mark, the check box is selected. When the square is blank, the check box is unselected. By default, a check box is initially unselected, and when a user clicks on a check box, it toggles between unselected and selected. For an example of an unselected check box, see the "I accept…" check box at the top of Figure 19.3.

To create a check box control, call

the CheckBox constructor like this:

CheckBox *check-box-reference* = new CheckBox(*label*);

The *label* argument specifies the text that appears at the right of the check box’s square. If the label argument is omitted, no text appears at the right of the check box’s square. Here’s how we created the check box in Figure 19.3’s “I accept…” check box:

```
CheckBox confirmBox = new CheckBox(
        "I accept the terms of this agreement.");
```

Here are the API headings and descriptions for some of the more popular CheckBox methods:

```
public boolean isSelected()
```

Returns true if the check box is selected and false otherwise.

public void setDisable(boolean flag)

Disables the check box if the parameter is true.

public void setOnAction(*event handler*)

Registers a method reference or a lambda expression.

public void setSelected(Boolean flag)

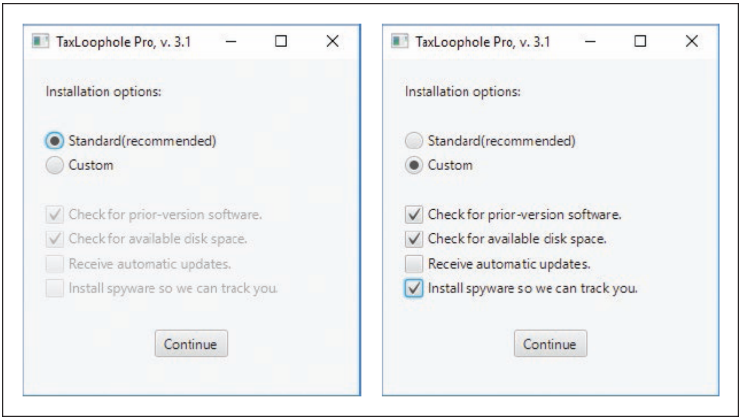
Makes the check box selected or unselected.

public void setVisible(Boolean flag)

Makes the check box visible or invisible.

The is Selected and setVisible methods are straightforward, and the setOnAction method works the same as with the Button control, but setSelected and setDisable deserve some attention. Why might you want to call setSelected and adjust the selection status of a check box?

Because you might want the program to control how one control's user input impacts another control. For example, in Figure 19.6, selecting the standard or custom radio button (we'll describe radio buttons in the next subsection) impacts the check box selections. More specifically, if the user selects the Standard option, the check box selections should go to their "standard" settings. As you can see in Figure 19.6's left window, the standard settings for the check boxes are the top two selected and the bottom two unselected. To have your program select the top two check boxes, have those two check boxes call setSelected(true). To have your program unselect the bottom two check boxes, have those two check boxes call setSelected(false).



TaxLoophole Pro, v. 3.1

Installation options:

● Standard(recommended)
○ Custom

☑ Check for prior-version software.
☑ Check for available disk space.
☐ Receive automatic updates.
☐ Install spyware so we can track you.

TaxLoophole Pro, v. 3.1

Installation options:

○ Standard(recommended)
● Custom

☑ Check for prior-version software.
☑ Check for available disk space.
☐ Receive automatic updates.
☑ Install spyware so we can track you.

Figure 19.6 Software installation window after the user selects the standard mode and then the custom mode

©JavaFX

To have your program disable a check box, have the checkbox call setDisable(true). Why might you want to disable a check box? Because you might want to keep the user from modifying that box's value. For example, if the user selects the Standard option as shown in Figure 19.6's left window, the program should not only select the top two check boxes (as explained earlier), but also it should disable all four check boxes by having each check box call setDisable(true). That way, the user cannot make changes to the standard-configuration check box values. In the left window, note that the check boxes' labels are lighter gray. That behavior comes automatically when a check box is disabled. If the user selects the Custom option as shown in Figure 19.6's right window, the program should enable all four check boxes by having each check box call setDisable(false).

With a Button control you'll almost always want associated event handling. But with a CheckBox control, you may or may not want associated event handling. A check box with no event handling simply serves as an input entity. In that case, the program reads and processes the check box's value (checked or unchecked) when the user clicks a button. On the other hand, if you want something to happen immediately (right when the user selects a check box), then add an event handler for the check box control. Suppose you have a Green Background check box. If you want the window's background color to change to green right when the user clicks the check box, then add an event handler to the check box that assigns green to the pane's -fx-background-color property. The syntax for adding an event handler to a check box is the same as for a button. Have the check box control call its setOnAction method with a lambda-expression argument or a method reference argument.

RadioButton Control

Now go back to Figure 19.6 and look at the radio buttons at the top of the window. A RadioButton control displays a small circle with a label to its right. When the circle contains a dot, the radio button is selected. When the circle is empty, the radio button is unselected.

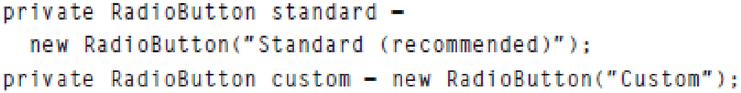
According to the description so far, radio buttons sound a lot like check boxes. They display a shape and a label, and they keep track of whether something is on or off. The key difference between radio buttons and check boxes is that radio buttons almost always come in *toggle groups*. Within a radio button toggle group, only one radio button can be selected at a time. If a user clicks an unselected radio button, the clicked button becomes selected, and if another button was previously selected in the group, it becomes unselected. If a user clicks a selected radio button, no change occurs (i.e., the clicked button remains selected). In contrast, if a user clicks a selected check box, the check box changes its state from selected to unselected.

To create a RadioButton control, call the RadioButton constructor like this:

RadioButton *radio-button-reference* = new RadioButton(*label*);

The *label* argument specifies the text that appears at the right of the radio button's circle. If the label argument is omitted, no text appears at the right of the radio button's circle. By default, a radio button is initially unselected.

The following statements show how we created the standard and custom radio buttons for the windows shown in Figure 19.6:



```
private RadioButton standard =
  new RadioButton("Standard (recommended)");
private RadioButton custom = new RadioButton("Custom");
```

To enable the only-one-button-selected-at-a-time radio-button-group functionality, create a ToggleGroup object and add individual radio button controls to it. Here's how:

first-button-in-group.setToggleGroup(*toggle-*

group-reference);

...

last-button-in-group.setToggleGroup(*toggle-group-reference*);

The following statements show how we create a radio button toggle group for the standard and custom radio buttons:

```
ToggleGroup radioGroup = new ToggleGroup();

...

standard.setToggleGroup(radioGrou
custom.setToggleGroup(radioGroup
```

In addition to adding radio buttons to a toggle group, you also must add them to a container. Radio buttons work the same as other components in terms of adding them to a container. You can add them individually, or you can add them along with other components, like this:

```
VBox panel = new VBox();
...
panel.getChildren().addAll(
  new Label("Installation options:"), new Label(),
  standard, custom, new Label(),
  prior, diskSpace, updates, spyware);
```

⚠ You need to add each radio button twice—once to a radio button group and once to a container. If you like shortcuts, you might be thinking: Why does Java make you add the individual radio buttons to the container? Why can't you just add the ToggleGroup object, which groups the radio buttons together, to the container? The technical answer is that ToggleGroup does not actually "contain" the radio buttons because it's not a descendant of the Parent class. In fact, it's not even a Node, so it can't be part of a scene. Besides, where would the ToggleGroup have them go? Adding the buttons separately gives you freedom to put them where you want. You can even put them in different panes.

As expected, the RadioButton class is defined in the javafx.scene.control package. As for the ToggleGroup class, it's also defined in the javafx.scene.control package, even though it's not a descendant of the Node class (very weird).

Here are some of the more popular RadioButton methods:

isSelected, setDisable, setOnAction, setSelected, setVisible

We described these same methods for the CheckBox control. Only one of them needs further attention—the setSelected method. To understand how setSelected works, you first need to understand fully how a user interacts with a radio button group. To select a radio button, a user clicks it. That causes the radio button to become selected and all other radio buttons in the group to become unselected. To programmatically select a radio button, you have the radio button call setSelected(true). That causes the radio button to become selected and all other radio buttons in the group to become unselected. As mentioned above, there is no way for a user to unselect a button. Likewise, there is no way for a program to unselect a button. That's why calling setSelected(false) doesn't do anything. It compiles and runs, but it doesn't cause any buttons to change their selected status.

Installation Program

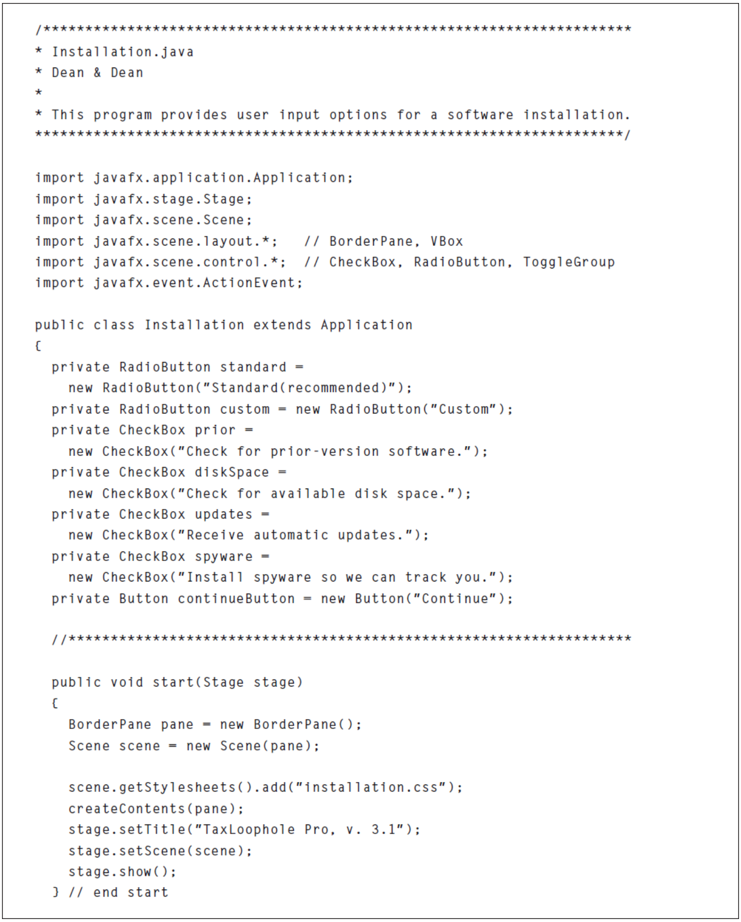
Take a look at the program in Figures 19.7a, 19.7b, and 19.7c. It's the Installation program, and it generates the windows we've been referring to in Figure 19.6. Figure 19.7a shows the imports, instance variables, and start method. We use instance variables for the check box and radio button controls because we access those entities in the program's two event handlers. That means their values need to persist; thus the need for instance variables.

Figure 19.7b's code includes the createContents method, which adds components to the window, and a setStandard method, which implements the event handler for when the user clicks the standard button. In the createContents method, notice that we use two containers—a BorderPane and a VBox. We want the Continue button to be centered at the bottom, and to achieve that placement, we use the center compartment of the BorderPane layout, and we do not use the bottom compartment at all. For the other components (label, radio button, and check box controls), we want them to be arranged vertically and left aligned, and for that placement, we use the VBox layout. Note in the createContents method how we add the VBox panel (a panel is another name for a pane) to the BorderPane's top compartment and then we add the controls to the VBox.

Continuing with our examination of the createContents method, you can see that we instantiate a ToggleGroup named radioGroup. We then assign the radio buttons to the radioGroup by calling setToggleGroup like this:

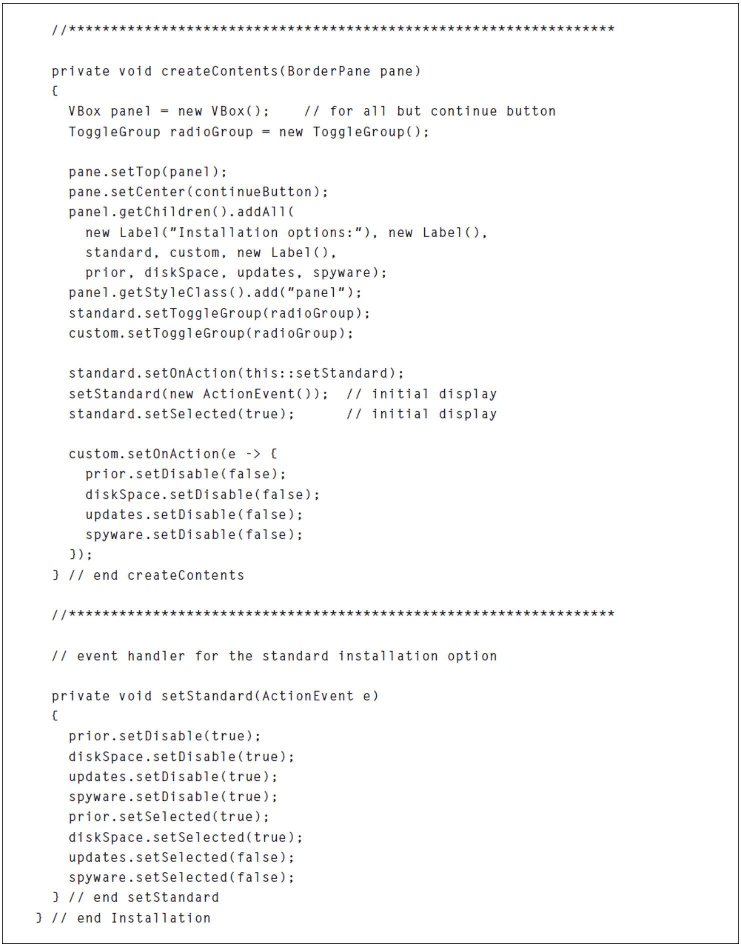
```
standard.setToggleGroup(radioGroup);
custom.setToggleGroup(radioGroup);
```

After that, the standard radio button and the custom radio button each call setOnAction to register their two event handlers. For the standard button we use a method reference, and for the custom button we use a lambda expression. Why the difference? The custom button's event handler is relatively short, so it's a good candidate for a lambda expression. It simply enables the check boxes. The standard button’s event handler is kinda long, and even more importantly, we need to call it from more than one place, so it’s a good candidate for a method reference. We call it every time the user clicks the standard button (of course), but we also call it initially when the window first displays. We call the standard button’s event handler initially so the first two text boxes get selected and all four text boxes get disabled.



```
/*********************************************************************
* Installation.java
* Dean & Dean
*
* This program provides user input options for a software installation.
*********************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;   // BorderPane, VBox
import javafx.scene.control.*;  // CheckBox, RadioButton, ToggleGroup
import javafx.event.ActionEvent;

public class Installation extends Application
{
  private RadioButton standard =
    new RadioButton("Standard(recommended)");
  private RadioButton custom = new RadioButton("Custom");
  private CheckBox prior =
    new CheckBox("Check for prior-version software.");
  private CheckBox diskSpace =
    new CheckBox("Check for available disk space.");
  private CheckBox updates =
    new CheckBox("Receive automatic updates.");
  private CheckBox spyware =
    new CheckBox("Install spyware so we can track you.");
  private Button continueButton = new Button("Continue");

  //******************************************************************

  public void start(Stage stage)
  {
    BorderPane pane = new BorderPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add("installation.css");
    createContents(pane);
    stage.setTitle("TaxLoophole Pro, v. 3.1");
    stage.setScene(scene);
    stage.show();
  } // end start
```

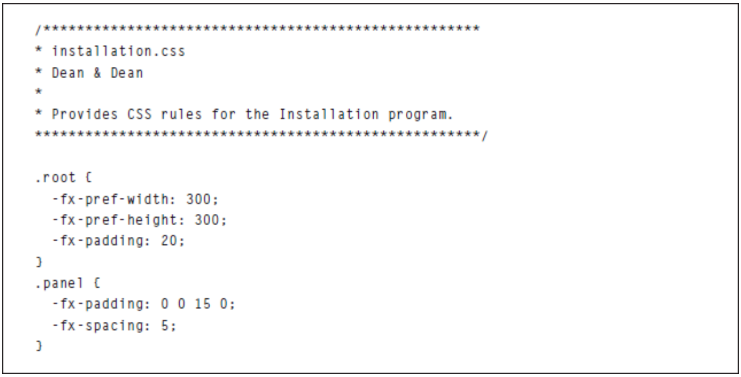
Figure 19.7a Installation program—part A



```
  //*****************************************************************

  private void createContents(BorderPane pane)
  {
    VBox panel = new VBox();    // for all but continue button
    ToggleGroup radioGroup = new ToggleGroup();

    pane.setTop(panel);
    pane.setCenter(continueButton);
    panel.getChildren().addAll(
      new Label("Installation options:"), new Label(),
      standard, custom, new Label(),
      prior, diskSpace, updates, spyware);
    panel.getStyleClass().add("panel");
    standard.setToggleGroup(radioGroup);
    custom.setToggleGroup(radioGroup);

    standard.setOnAction(this::setStandard);
    setStandard(new ActionEvent());  // initial display
    standard.setSelected(true);      // initial display

    custom.setOnAction(e -> {
      prior.setDisable(false);
      diskSpace.setDisable(false);
      updates.setDisable(false);
      spyware.setDisable(false);
    });
  } // end createContents

  //*****************************************************************

  // event handler for the standard installation option

  private void setStandard(ActionEvent e)
  {
    prior.setDisable(true);
    diskSpace.setDisable(true);
    updates.setDisable(true);
    spyware.setDisable(true);
    prior.setSelected(true);
    diskSpace.setSelected(true);
    updates.setSelected(false);
    spyware.setSelected(false);
  } // end setStandard
} // end Installation
```

Figure 19.7b Installation program—part B

Figure 19.7c shows the Installation program's style sheet. The CSS rules' width, height, padding, and spacing property-value pairs should be self-explanatory. There's no need for alignment-oriented property-value pairs because the alignments come from the BorderPane's default behavior. Specifically, the label, check box, and radio button controls are left aligned due to their being in the BorderPane's top compartment. The button is center aligned due to its being in the BorderPane's center compartment.



```
/*****************************************************
* installation.css
* Dean & Dean
*
* Provides CSS rules for the Installation program.
*****************************************************/

.root {
  -fx-pref-width: 300;
  -fx-pref-height: 300;
  -fx-padding: 20;
}
.panel {
  -fx-padding: 0 0 15 0;
  -fx-spacing: 5;
}
```

Figure 19.7c Style sheet for Installation program

19.5 ComboBox Control

User Interface

A *combo box* allows a user to select an item from a list of items. Combo boxes are sometimes called *drop-down lists* because if a user clicks the control's down arrow, a list of selection items drops down from the original box. Figure 19.8 shows a label and a combo box stored in an HBox. You've already learned about the label control and the HBox container, so we'll focus on the combo box.

When a combo box first displays, it's empty by default, meaning there's no initial selection. After the user clicks the down arrow and makes a selection from the item list, the list contracts and the selected item displays in the box. Combo boxes are similar to radio button groups in that they both allow the user to select one item from a list of items. But a combo box takes up less space in the window. Therefore, if you have a long list of items to choose from, and you want to save space, you should use a combo box rather than a group of radio buttons. By default, a combo box's item list displays no more than 10 items at a time. If the list contains more than 10 items, a vertical scroll bar appears. This allows users to find any item in a long list.

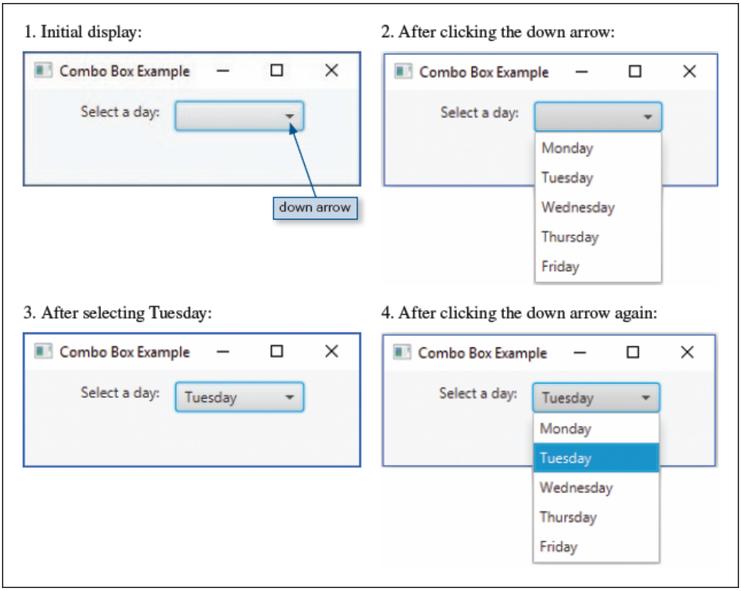

1. Initial display:
Combo Box Example
Select a day:
down arrow
2. After clicking the down arrow:
Combo Box Example
Select a day:
Monday
Tuesday
Wednesday
Thursday
Friday
3. After selecting Tuesday:
Combo Box Example
Select a day:
Tuesday
4. After clicking the down arrow again:
Combo Box Example
Select a day:
Tuesday
Monday
Tuesday
Wednesday
Thursday
Friday

Figure 19.8 How a ComboBox works

©JavaFX

Implementation

As with the other controls, the

ComboBox's class is defined in the javafx.scene.control package. Creating a combo box is a two-step process. First, instantiate it using this syntax:

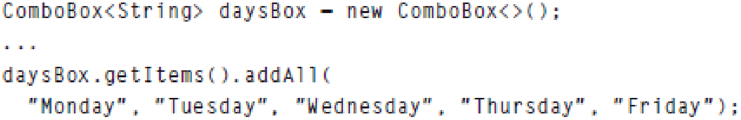
ComboBox<*reference-type*> *combo-box-reference* = new ComboBox< >();

Note the <*reference-type*> in the above syntax. You might recall that the angled brackets (< >) are indicative of a generic class. As such, when you declare a ComboBox, you are required to specify the type for the items that comprise the combo box's item list. That type must be a reference type, not a primitive type. For a combo box, that reference type will almost always be the String class.

To populate the box with a list of items, use this syntax:

combo-box-reference.getItems().addAll(*comma-separated-list-of-items*);

The items in the item list must be references of the type specified by the combo box's declaration (which means strings, for the most part). Note the day-of-the-week strings in the following example, which shows the code used to generate the combo box in Figure 19.8:



```
ComboBox<String> daysBox = new ComboBox<>();
...
daysBox.getItems().addAll(
  "Monday", "Tuesday", "Wednesday", "Thursday", "Friday");
```

Methods

Here are some of the more popular ComboBox methods:

getValue, setDisable, setEditable, setOnAction, setValue, setVisible

The setDisable, setOnAction, and setVisible methods should look familiar; they work the same for the ComboBox control as they do for the other controls. Here are the API headings and descriptions for the other three methods:

public T getValue()

> Returns the combo box's selected item. If the combo box is editable and the user enters a value, then that value is the selected item that's returned.

public void setEditable(boolean flag)

> Makes the combo box's top portion editable or noneditable.

public void setValue(T item)

> Selects a specified item from the combo box's item list and displays that selected item at the top of the combo box.

The get Value method provides a way for the program to retrieve the selected item from a combo box. Typically, you'd call getValue from a button's event handler that's in charge of processing the user's input values.

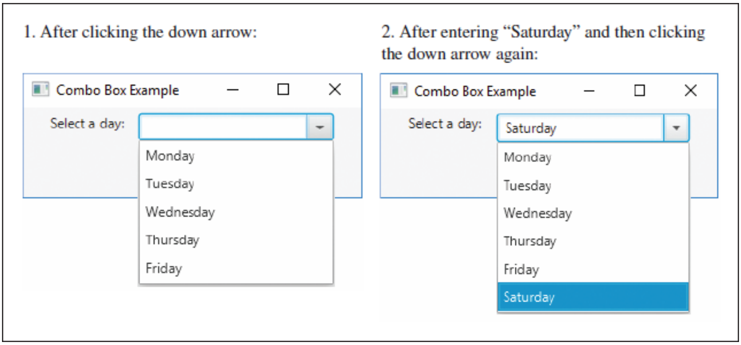
By default, a user cannot modify a combo box's item list. However, a setEditable(true) method call enables the user to make entries in the top portion of the combo box. In that case, the top portion looks and acts like a TextField control. A user entry in an editable ComboBox automatically becomes the ComboBox's selected item. A subsequent getValue method call retrieves that entered value.

The setValue method provides a way for the program to select an item. You can use it to initialize the selection when the program first loads. The type of that item must match the item type when the ComboBox is declared. If the argument passed to the setValue method is not already in the item list, no problem; it's automatically added to the list, and it becomes the selected item.

Programmatically Manipulating a ComboBox

Suppose you'd like to enable the user to add another day of the week to the day-of-the-week combo box shown earlier. See Figure 19.9 for what we're talking about—after the user types "Saturday" and presses enter, "Saturday" is added to the combo box's item list.

To implement that functionality, you'll need to make the combo box editable and register an event handler with the combo box. The event handler fires when the user presses enter within the combo box's editable area. It retrieves the user's entry and assigns that entry to the combo box's item list. Assuming daysBox is the name of the combo box, here's the code that does what we're talking about:



1. After clicking the down arrow:

2. After entering "Saturday" and then clicking the down arrow again:

Combo Box Example
Select a day:
Monday
Tuesday
Wednesday
Thursday
Friday
Combo Box Example
Select a day:
Saturday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

Figure 19.9 How an editable ComboBox works

©JavaFX

```
daysBox.setEditable(true);
daysBox.setOnAction(e -> {
  String selection = daysBox.getValue();

  if (!daysBox.getItems().contains(selection)) // avoid duplication
  {
    daysBox.getItems().add(selection);
  }
});
```

In the code fragment, note the if statement's heading. It calls the getItems and contains methods to determine whether the user's entry is already in the list. If not, then the entry gets added to the list by calling the getItems and add methods.

The ComboBox's getItems method returns the combo box's item list, but more formally, that list is an ObservableList interface. An

ObservableList is a list that enables event handlers to monitor changes in the list when they occur. You've already learned about two methods from the ObservableList interface—contains and add. Now for a few more….

If you'd like to remove one item, like "Tuesday," from daysBox's item list, use a statement like this:

```
daysBox.getItems().remove("Tuesday");
```

If you'd like to remove all items from the list and start over, use a statement like this:

```
daysBox.getItems().clear();
```

The ObservableList's indexOf method receives an item and returns the index of where that item is within the item list. The get method goes the other direction. It receives an index value and returns the item in the item list at that index position. For example, you might want to find the item that is immediately after the currently selected item. To do that you could use a code fragment like this:

```
String selection = daysBox.getValue();
int index = daysBox.getItems().indexOf(selection);
String item = daysBox.getItems().get(index + 1);
```

19.6 Job Application Program

In this section, we'll put into practice what you've learned in the last two sections. We present a complete program that uses check boxes, radio buttons, and a combo box. The program implements a job application form. If the user enters values that are indicative of a good employee, the program generates a secondary window and displays an encouraging message (“Thank you for your application submission. We’ll contact you after we process your information.”) in it. If the user enters information indicative of a bad employee, the program generates a secondary window and displays a discouraging message in it. See Figure 19.10 to get a better idea of what’s going on.

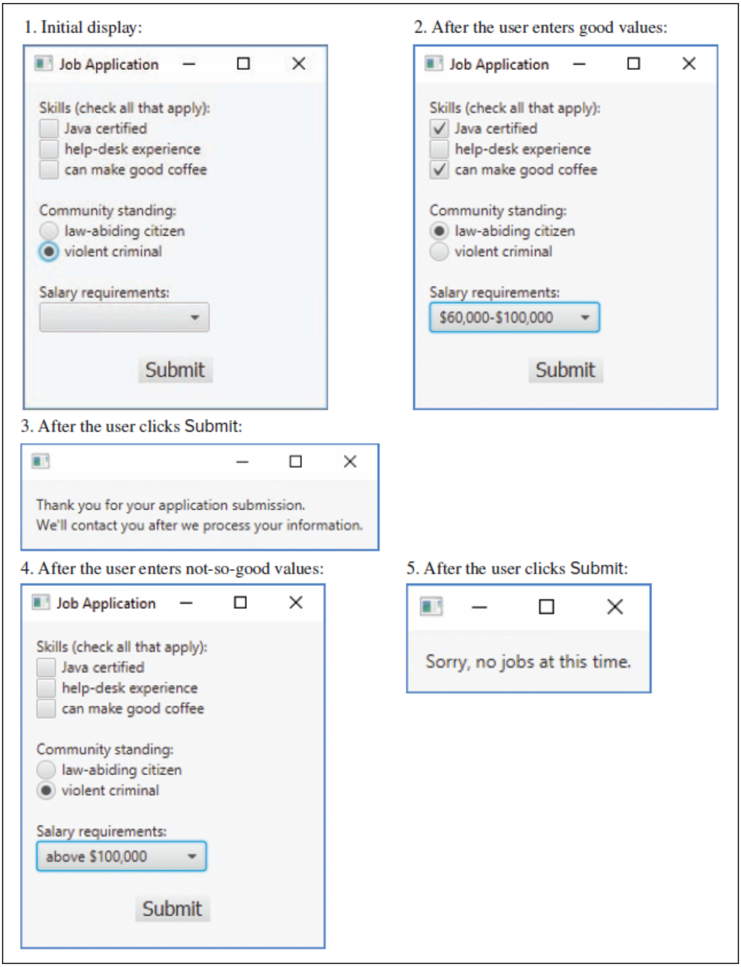
The JobApplication program is kinda long, and it spans three figures—19.11a, 19.11b, and 19.11c. In Figure 19.11a notice how the program provides complete descriptions of all seven of the active components right at the beginning. These are all instance variables because the program needs to access each of them during the set-up phase and again later in the event-handling method. Defining these components completely at the beginning also provides nice self-documentation and makes the program easier to understand.

For the window's layout we use a BorderPane with a VBox embedded in the top area and a submit button in the center area. The VBox holds all the components except the button. By default, a BorderPane left-aligns the contents of its top area and centers the contents of its center area. By default, a VBox left-aligns its contents. Thus, as you can see in Figure 19.10, the JobApplication window displays all the VBox components with left alignment. As for the button, it's positioned in the center area, so the JobApplication program displays it with center alignment.

Figure 19.11b shows the createContents method. First it connects a JavaFX CSS rule named pane to the primary window's BorderPane, also named pane. Then it populates the BorderPane's top and center areas with a VBox and a submit button, respectively. Then it adds all the other components to the VBox, using two blank Label components to create blank lines between the three different types of information. Then it ties the two radio buttons to a toggle group and adds the three salary ranges to the ComboBox. The next two statements connect JavaFX CSS rules to the VBox panel and the submit button. We added those rules at the end to make things look good. Get used to that sort of thing—there's a lot of interface tweaking with GUI programming. The final statement in createContents is a setOnAction method call that registers a lambda expression event handler with the submit button.

Let's now examine the submit button's event handler. It first verifies that the user has made some kind of selection in the last two categories—

community standing and salary requirements. Then it decides whether to issue a favorable response or an unfavorable response. For a favorable response, the user must have selected the law-abiding citizen radio button and must have selected less than $100,000 in salary. For each of the three possibilities—insufficient information, favorable response, or unfavorable response—code inside the lambda expression calls another helper method, showResponse. Each method calls passes a string argument that specifies the message for the displayed response.



1. Initial display:

Skills (check all that apply):
☐ Java certified
☐ help-desk experience
☐ can make good coffee

Community standing:
○ law-abiding citizen
● violent criminal

Salary requirements:

2. After the user enters good values:

Skills (check all that apply):
☑ Java certified
☐ help-desk experience
☑ can make good coffee

Community standing:
● law-abiding citizen
○ violent criminal

Salary requirements:
$60,000-$100,000

3. After the user clicks Submit:

Thank you for your application submission.
We'll contact you after we process your information.

4. After the user enters not-so-good values:

Skills (check all that apply):
☐ Java certified
☐ help-desk experience
☐ can make good coffee

Community standing:
○ law-abiding citizen
● violent criminal

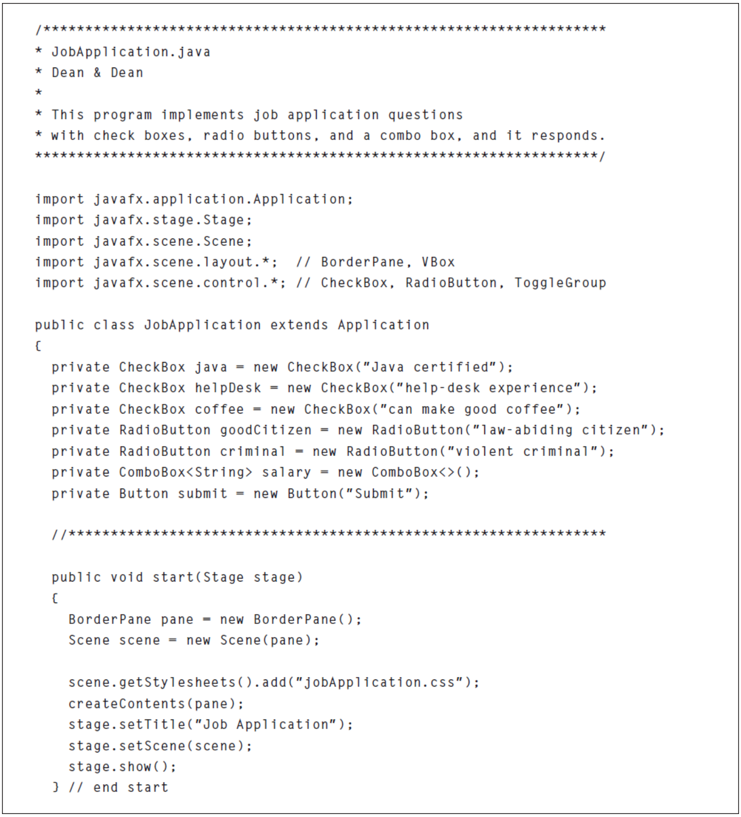
Salary requirements:
above $100,000

5. After the user clicks Submit:

Sorry, no jobs at this time.

Figure 19.10 Sample session for the JobApplication

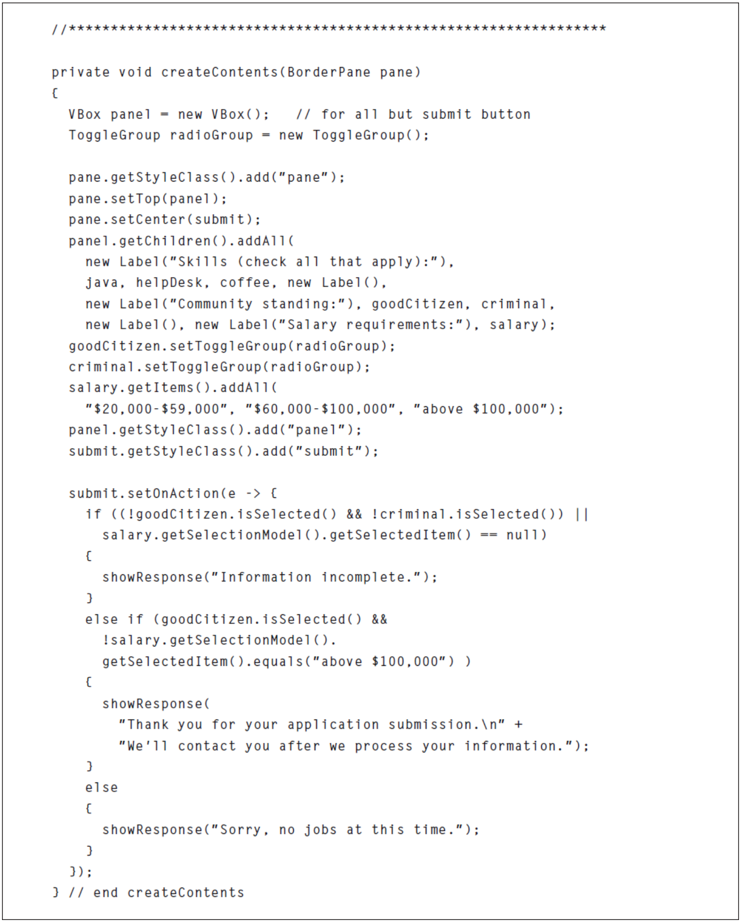
©JavaFX



```
/*******************************************************************
* JobApplication.java
* Dean & Dean
*
* This program implements job application questions
* with check boxes, radio buttons, and a combo box, and it responds.
*******************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;  // BorderPane, VBox
import javafx.scene.control.*; // CheckBox, RadioButton, ToggleGroup

public class JobApplication extends Application
{
  private CheckBox java = new CheckBox(”Java certified”);
  private CheckBox helpDesk = new CheckBox(”help-desk experience”);
  private CheckBox coffee = new CheckBox(”can make good coffee”);
  private RadioButton goodCitizen = new RadioButton(”law-abiding citizen”);
  private RadioButton criminal = new RadioButton(”violent criminal”);
  private ComboBox<String> salary = new ComboBox<>();
  private Button submit = new Button(”Submit”);

  //*****************************************************************

  public void start(Stage stage)
  {
    BorderPane pane = new BorderPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add(”jobApplication.css”);
    createContents(pane);
    stage.setTitle(”Job Application”);
    stage.setScene(scene);
    stage.show();
  } // end start
```

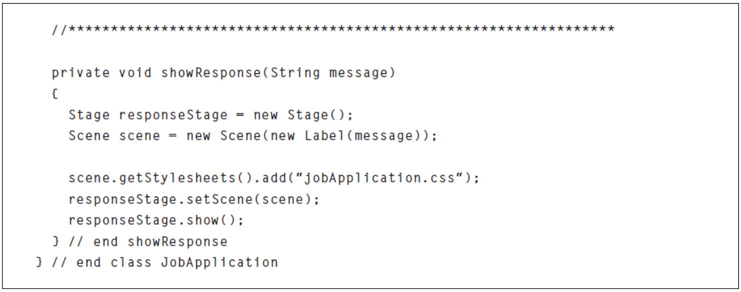
Figure 19.11a JobApplication program—part A

Figure 19.11c contains the showResponse method. First off, the method instantiates a new Stage object for the secondary window that displays the response. Then it instantiates a Scene with a Label containing the passed-in message parameter. Next it registers the same style sheet used for the primary window. Using the same style sheet helps with maintenance and visual consistency.



```
//*******************************************************************

private void createContents(BorderPane pane)
{
  VBox panel = new VBox();   // for all but submit button
  ToggleGroup radioGroup = new ToggleGroup();

  pane.getStyleClass().add("pane");
  pane.setTop(panel);
  pane.setCenter(submit);
  panel.getChildren().addAll(
    new Label("Skills (check all that apply):"),
    java, helpDesk, coffee, new Label(),
    new Label("Community standing:"), goodCitizen, criminal,
    new Label(), new Label("Salary requirements:"), salary);
  goodCitizen.setToggleGroup(radioGroup);
  criminal.setToggleGroup(radioGroup);
  salary.getItems().addAll(
    "$20,000-$59,000", "$60,000-$100,000", "above $100,000");
  panel.getStyleClass().add("panel");
  submit.getStyleClass().add("submit");

  submit.setOnAction(e -> {
    if ((!goodCitizen.isSelected() && !criminal.isSelected()) ||
      salary.getSelectionModel().getSelectedItem() == null)
    {
      showResponse("Information incomplete.");
    }
    else if (goodCitizen.isSelected() &&
      !salary.getSelectionModel().
      getSelectedItem().equals("above $100,000") )
    {
      showResponse(
        "Thank you for your application submission.\n" +
        "We'll contact you after we process your information.");
    }
    else
    {
      showResponse("Sorry, no jobs at this time.");
    }
  });
} // end createContents
```

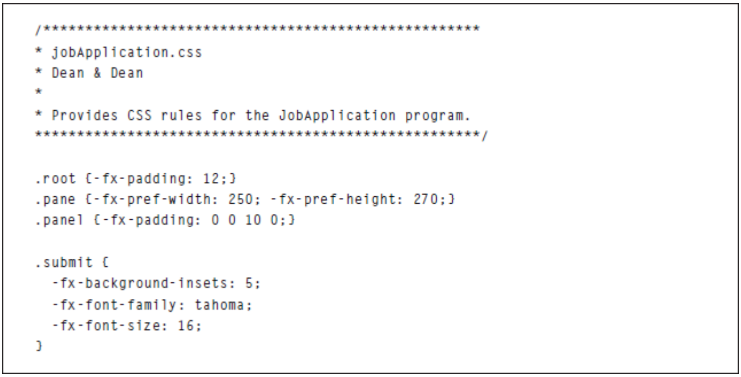
Figure 19.11b JobApplication program —part B



```
  //*******************************************************************

  private void showResponse(String message)
  {
    Stage responseStage = new Stage();
    Scene scene = new Scene(new Label(message));

    scene.getStylesheets().add("jobApplication.css");
    responseStage.setScene(scene);
    responseStage.show();
  } // end showResponse
} // end class JobApplication
```

Figure 19.11c JobApplication program—part C

Figure 19.12 shows the style sheet that specifies the JavaFX CSS rules for the JobApplication program. The .root rule applies to every scene graph that employs the style sheet, so both windows display with a padding of 12 pixels. The .pane rule gets applied to the primary window (as you can verify in the program's createContents method), so the primary window displays with the specified 250 × 270 pixel dimensions. The secondary window, with no explicit JavaFX CSS rule for its width and height, displays with dimensions determined by its content. The .panel rule gets applied to the primary window’s VBox pane, which holds all the components other than the button. Its -fx-padding: 0 0 10 0; property-value pair assigns 10 pixels of padding along the bottom edge of the VBox in order to provide additional separation between the VBox’s components and the button. The .submit rule gets applied to the primary window’s button. Its font property-value pairs are straightforward. Its -fx-background-insets: 5; property-value pair inserts whitespace outside of the button, which causes the button’s border to shrink so it’s closer to the button’s label.



```
/******************************************************
* jobApplication.css
* Dean & Dean
*
* Provides CSS rules for the JobApplication program.
******************************************************/

.root {-fx-padding: 12;}
.pane {-fx-pref-width: 250; -fx-pref-height: 270;}
.panel {-fx-padding: 0 0 10 0;}

.submit {
  -fx-background-insets: 5;
  -fx-font-family: tahoma;
  -fx-font-size: 16;
}
```

Figure 19.12 Style sheet for

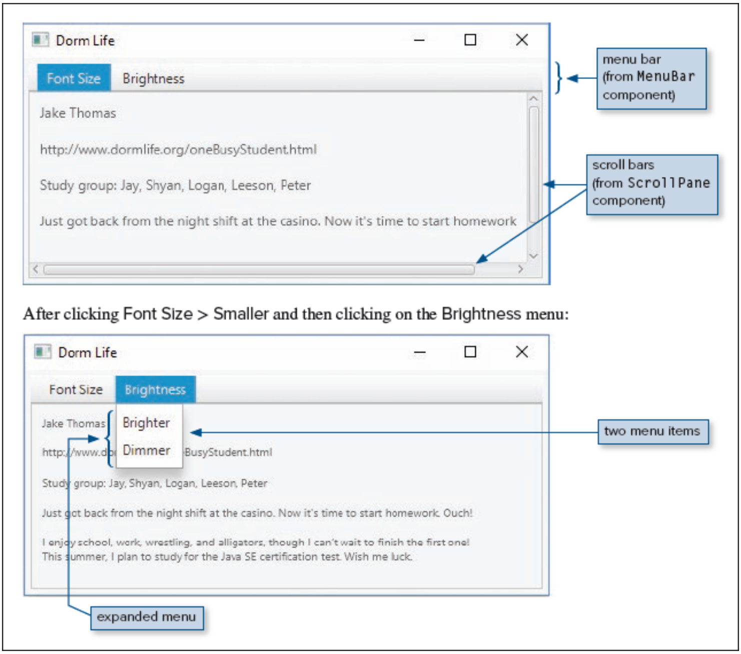
JobApplication program

19.7 ScrollPane and Menu Classes

In this section, we describe the ScrollPane component and the various components that comprise a menuing system. We use those components as part of a program that enables the user to select customized formatting for the program's window. The program performs the customization by calling methods within an event handler that update JavaFX CSS properties.

Figure 19.13 shows the window generated by the upcoming AdjustableReader program. The program implements a simple reader application. In reading the window's text, the user can scroll horizontally and vertically and can use the Font

Size and Brightness menus to adjust the displayed text's size and brightness, respectively.


Dorm Life
Font Size
Brightness
menu bar (from MenuBar component)
scroll bars (from ScrollPane component)

Jake Thomas

http://www.dormlife.org/oneBusyStudent.html

Study group: Jay, Shyan, Logan, Leeson, Peter

Just got back from the night shift at the casino. Now it's time to start homework

After clicking Font Size > Smaller and then clicking on the Brightness menu:

Dorm Life
Font Size
Brightness
Brighter
Dimmer
two menu items
expanded menu

Jake Thomas

http://www.d BusyStudent.html

Study group: Jay, Shyan, Logan, Leeson, Peter

Just got back from the night shift at the casino. Now it's time to start homework. Ouch!

I enjoy school, work, wrestling, and alligators, though I can't wait to finish the first one!
This summer, I plan to study for the Java SE certification test. Wish me luck.

Figure 19.13 A reader program window that uses a scroll pane and a menu bar

©JavaFX

Note how the AdjustableReader window has two scroll bars. They allow the user to scroll up and down and sideways to reveal the content that is out of the current view. The user can also move the pane's content by *panning* with a mouse (panning is achieved by holding the mouse button down on the text and dragging). That scrolling and panning functionality comes courtesy of the window's ScrollPane component. A ScrollPane is a scrollable container that displays a node within it. In the AdjustableReader program, the ScrollPane's contained node is a Text object, which doesn't do much other than serve as a simple repository for text in a Node component. By default, a scroll pane provides zero, one, or two scroll bars as the content requires. Or a program can explicitly display one or both of the scroll bars. As a programmer, you can configure a scroll pane in various ways, such as enabling or disabling panning.

The AdjustableReader window's menus (for font size and brightness) are implemented with the MenuBar, Menu, and MenuItem classes. As you'll see when we examine the AdjustableReader code, to create menus, you add a MenuBar object to the window's scroll pane, then you add Menu objects to the MenuBar object, and finally you add MenuItem objects to each of the Menu objects.

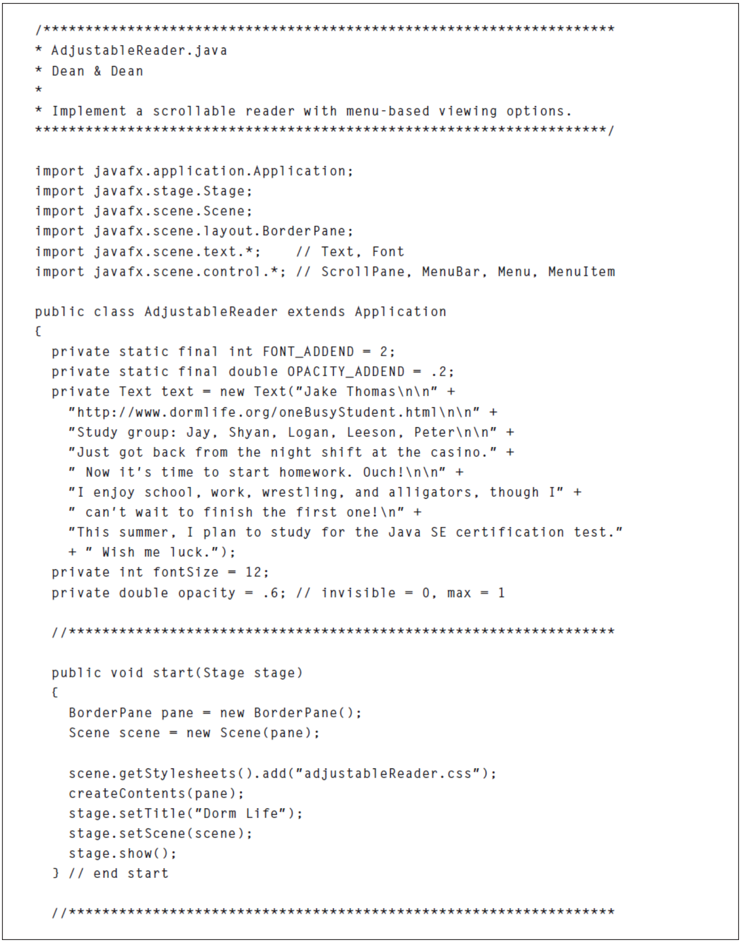
AdjustableReader Program Implementation Details

Figure 19.14a shows the first part of the AdjustableReader program. We'll skip the code that looks familiar and focus on the new stuff. Note how we declare a text instance variable with the Text class and initialize it with a huge string as part of its constructor call. We could have used a Label component instead of a Text component, but Label components are considered to be controls (the Label class is a descendant of the Control class), and Text objects are considered to be shapes (the Text class is a descendant of the Shape class). As such, Label components are supposed to be used as part of user input (e.g., as a prompt for a text box), whereas Text components are not supposed to be used as part of user input. For the -AdjustableReader program, the scroll pane displays text, and there's no user input, so using a Text component is appropriate.

Note the fontSize instance variable. We initialize it to 12, for a 12-point font size. As you'll see later on when we get to the event handler code, when the user selects the Larger font size option, we add FONT_ADDEND to fontSize, and when the user selects the smaller font size option, we subtract FONT_ADDEND from fontSize. Note how we declare FONT_ADDEND as an instance constant with a value of 2.

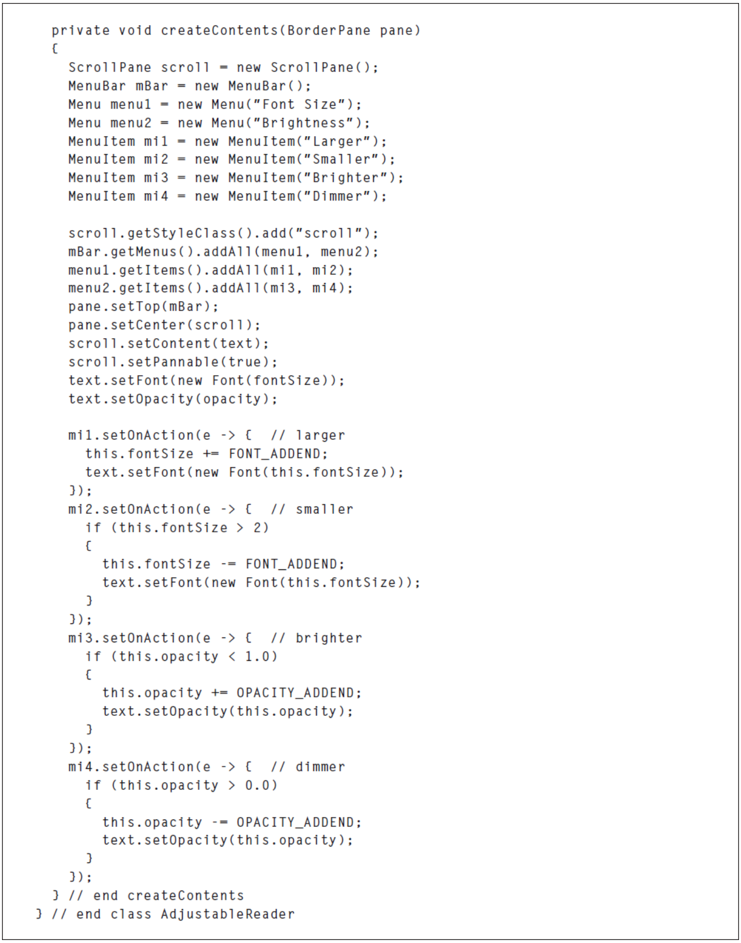
The logic for adjusting the text's brightness parallels the logic for adjusting the text's font size. Once again, we have an instance variable, an instance constant, and event handlers. The only tricky part is understanding what brightness means. We use the term "brightness" because that's the name of the menu, and that's a term that typical users will understand. But there's no brightness JavaFX CSS property. As you learned in Chapter 17, opacity refers to how opaque a color is. We use the opacity property (via the setOpacity method, inherited from the Node class) to adjust the text's brightness—greater opacity leads to greater brightness because the text's natural dark color contrasts with the window's natural white background. Note how we initialize the opacity instance variable to .6, where 1 is maximum opacity and 0 is complete transparency. As you'll see later on when we get to the event handler code, when the user selects the Brighter option, we add OPACITY_ADDEND to opacity, and when the user selects the Dimmer option, we subtract OPACITY_ADDEND from opacity. Note how we declare OPACITY_ADDEND as an instance constant with a value of .2.

Figure 19.14b contains the rest of the program—the createContents method. At the top of the createContents method, we instantiate a ScrollPane object and a MenuBar object, and all the component parts that comprise the menu system. We add the two menus to the menu bar, and then we add menu items to each of the two menus. The createContents method heading has one parameter, a BorderPane container which serves as the root for the program's scene graph. We add the menu bar to the BorderPane's top area and the scroll pane to the BorderPane's center area. Next, we set the scroll pane's pannability (not a word, but it should be), and we set the Text component's font size and opacity.



```
/*********************************************************************
* AdjustableReader.java
* Dean & Dean
*
* Implement a scrollable reader with menu-based viewing options.
*********************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;
import javafx.scene.text.*;    // Text, Font
import javafx.scene.control.*; // ScrollPane, MenuBar, Menu, MenuItem

public class AdjustableReader extends Application
{
  private static final int FONT_ADDEND = 2;
  private static final double OPACITY_ADDEND = .2;
  private Text text = new Text(”Jake Thomas\n\n” +
    ”http://www.dormlife.org/oneBusyStudent.html\n\n” +
    ”Study group: Jay, Shyan, Logan, Leeson, Peter\n\n” +
    ”Just got back from the night shift at the casino.” +
    ” Now it's time to start homework. Ouch!\n\n” +
    ”I enjoy school, work, wrestling, and alligators, though I” +
    ” can't wait to finish the first one!\n” +
    ”This summer, I plan to study for the Java SE certification test.”
    + ” Wish me luck.”);
  private int fontSize = 12;
  private double opacity = .6; // invisible = 0, max = 1

  //*******************************************************************

  public void start(Stage stage)
  {
    BorderPane pane = new BorderPane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add(”adjustableReader.css”);
    createContents(pane);
    stage.setTitle(”Dorm Life”);
    stage.setScene(scene);
    stage.show();
  } // end start

  //*******************************************************************
```

Figure 19.14a AdjustableReader program—part A



```
  private void createContents(BorderPane pane)
  {
    ScrollPane scroll = new ScrollPane();
    MenuBar mBar = new MenuBar();
    Menu menu1 = new Menu("Font Size");
    Menu menu2 = new Menu("Brightness");
    MenuItem mi1 = new MenuItem("Larger");
    MenuItem mi2 = new MenuItem("Smaller");
    MenuItem mi3 = new MenuItem("Brighter");
    MenuItem mi4 = new MenuItem("Dimmer");

    scroll.getStyleClass().add("scroll");
    mBar.getMenus().addAll(menu1, menu2);
    menu1.getItems().addAll(mi1, mi2);
    menu2.getItems().addAll(mi3, mi4);
    pane.setTop(mBar);
    pane.setCenter(scroll);
    scroll.setContent(text);
    scroll.setPannable(true);
    text.setFont(new Font(fontSize));
    text.setOpacity(opacity);

    mi1.setOnAction(e -> {  // larger
      this.fontSize += FONT_ADDEND;
      text.setFont(new Font(this.fontSize));
    });
    mi2.setOnAction(e -> {  // smaller
      if (this.fontSize > 2)
      {
        this.fontSize -= FONT_ADDEND;
        text.setFont(new Font(this.fontSize));
      }
    });
    mi3.setOnAction(e -> {  // brighter
      if (this.opacity < 1.0)
      {
        this.opacity += OPACITY_ADDEND;
        text.setOpacity(this.opacity);
      }
    });
    mi4.setOnAction(e -> {  // dimmer
      if (this.opacity > 0.0)
      {
        this.opacity -= OPACITY_ADDEND;
        text.setOpacity(this.opacity);
      }
    });
  } // end createContents
} // end class AdjustableReader
```

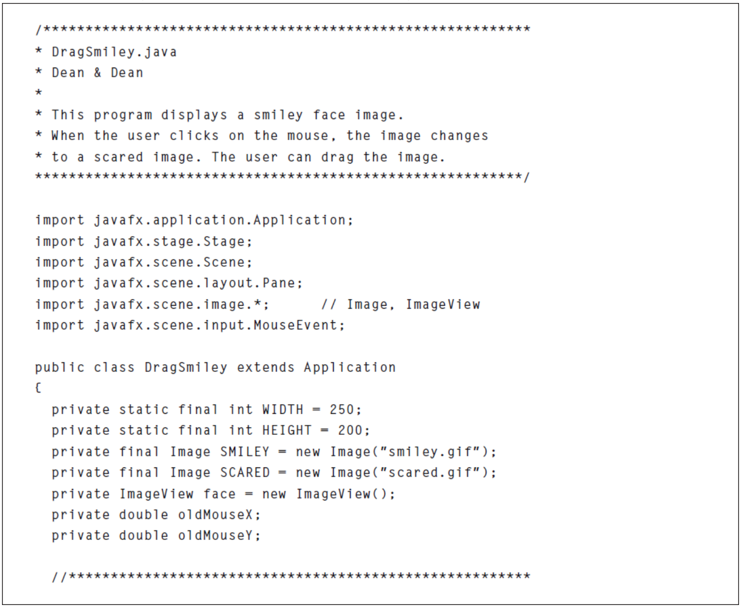
Figure 19.14b AdjustableReader program—part B

Page 924

The bottom half of the createContents method is devoted to registering event handlers with the four menu items. For the smaller font size option, we decrement the font size only if the current font size is greater than 2. That prevents dropping the font size down to zero or a negative value. Similar logic is used for opacity to keep the value between 0 and 1. Note how we adjust the Text component's font size—by calling the setFont method. Likewise, we adjust the Text component's opacity by calling the setOpacity method.

For its initial formatting, the AdjustableReader program uses the style sheet shown in Figure 19.15. The style sheet's rules are pretty straightforward—they set the window's dimensions and set padding for the window and the scroll pane. Remember the cascading part of CSS? If you refer back to Figure 17.14, you'll see that style sheet rules have a higher priority than component property method calls. The setFont and setOpacity method calls are component property method calls, so our

AdjustableReader program uses both techniques. So does that mean the setFont and setOpacity method calls are ignored? Nope. The style sheet’s rules take charge when the window first displays, but the setFont and setOpacity method calls win out when they’re executed as part of their event handlers. If you’re curious, copy those method calls to elsewhere in the program (outside of the event handlers), and you’ll see that they are ignored because they have lower priority than the style sheet’s rules.



```
/***********************************************************
* DragSmiley.java
* Dean & Dean
*
* This program displays a smiley face image.
* When the user clicks on the mouse, the image changes
* to a scared image. The user can drag the image.
***********************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.image.*;      // Image, ImageView
import javafx.scene.input.MouseEvent;

public class DragSmiley extends Application
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 200;
  private final Image SMILEY = new Image(”smiley.gif”);
  private final Image SCARED = new Image(”scared.gif”);
  private ImageView face = new ImageView();
  private double oldMouseX;
  private double oldMouseY;

  //*******************************************************
```

Figure 19.15 Style sheet for AdjustableReader program

19.8 Images and Mouse Events

We’ve already established that users expect GUI programs to look good, more so than for console-based programs. Previously, we talked about how to do that with color and fancy controls; now we talk about how to do that with images. Specifically, we describe how to use the Image class and how to manipulate images with event handlers.

The Image class is not a subclass of the Node class. Therefore, an Image instance cannot be a node in a scene graph. The workaround is to store an Image object in an ImageView object and then assign the ImageView object to the scene graph. That works because the ImageView class is a subclass of the Node class. We’re able to store an Image object in an ImageView object by having the ImageView object call its setImage method with an Image object as its argument.

We’ll illustrate this ImageView/ Image mechanism with the help of a DragSmiley program. It uses two images, a smiley-face image and a scared-face image. As indicated in Figure 19.16’s sample session, the program initially displays a smiley face in the top-left corner of the program’s window. If the user presses the left mouse button down on the image, the smiley image changes to the scared image (presumably because the smiley is apprehensive about what the user might do to it). If the user attempts to drag the image with the mouse, the drag works and the image displays as a scared face. When the user releases the button, the image changes back to a smiley face.

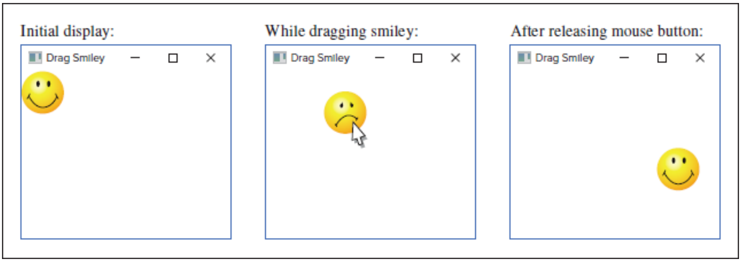

Initial display:
Drag Smiley
While dragging smiley:
Drag Smiley
After releasing mouse button:
Drag Smiley

Figure 19.16 Sample session for the DragSmiley program

©JavaFX

Implementing the DragSmiley Images

Figure 19.17a shows the first part of the DragSmiley program. In analyzing the program, let's start with the image code. Here are the named constants for the Image objects:

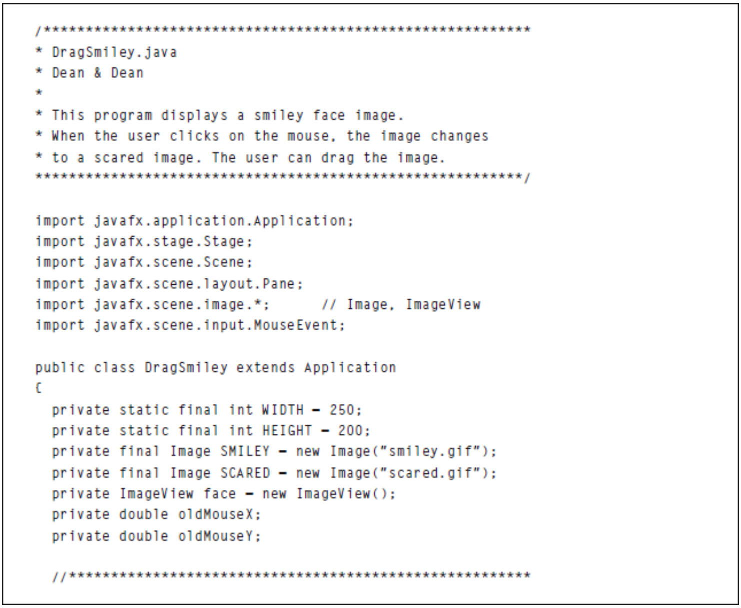
```
private final Image SMILEY = new Image("smiley.gif");
private final Image SCARED = new Image("scared.gif");
```

The Image constructor creates an

image object from its passed-in filename parameter. So in the above code fragment, two image objects are created from the smiley.gif and scared.gif files, respectively.1

The image objects eventually get stored in an ImageView object named face. Here's the code that instantiates the face object:

private ImageView face = new ImageView();



```
/**********************************************************
* DragSmiley.java
* Dean & Dean
*
* This program displays a smiley face image.
* When the user clicks on the mouse, the image changes
* to a scared image. The user can drag the image.
**********************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.image.*;      // Image, ImageView
import javafx.scene.input.MouseEvent;

public class DragSmiley extends Application
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 200;
  private final Image SMILEY = new Image("smiley.gif");
  private final Image SCARED = new Image("scared.gif");
  private ImageView face = new ImageView();
  private double oldMouseX;
  private double oldMouseY;

  //*******************************************************
```

Figure 19.17a DragSmiley program—part A

Here's how we add the face image object to the scene graph:

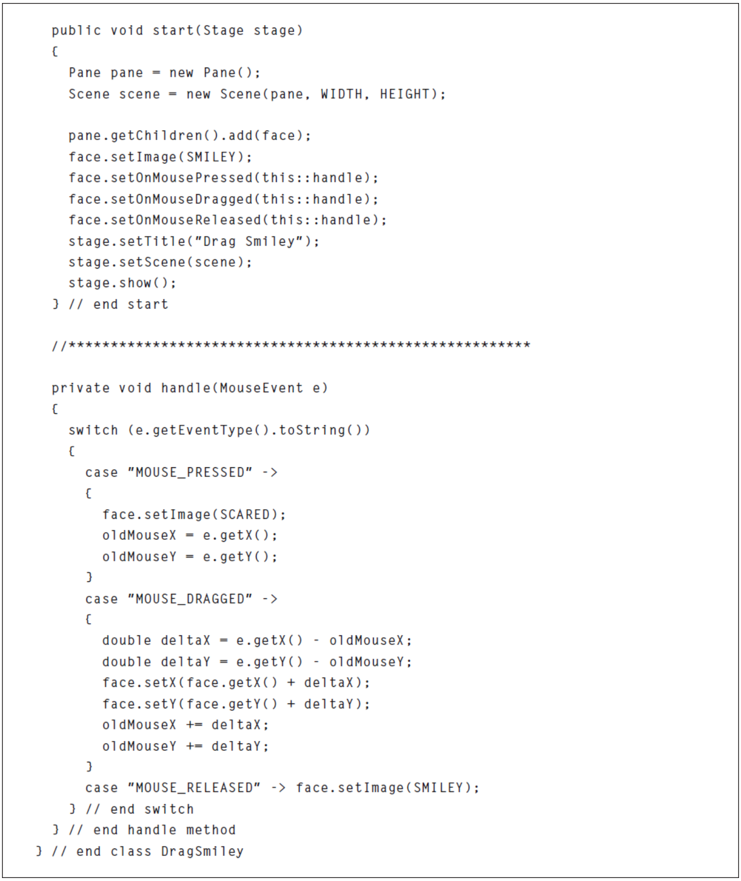
```
pane.getChildren().add(face);
```

And finally, here's how we add the initial smiley image to face:

```
face.setImage(SMILEY);
```

DragSmiley Event Handling with getEventType

Figure 19.17b shows the rest of the DragSmiley program—its event handler code. In our previous programs, we used an ActionEvent parameter to retrieve the component responsible for an event being fired (e.g., we used it to retrieve a text box when Enter was pressed and to retrieve a button when the button was clicked). The ActionEvent class isn't all that granular in terms of providing event details. For the DragSmiley program, we need more details, such as the position of the mouse when the user clicks it. Fortunately, there are lots of other event classes that are useful for different activities, and they tend to be more discriminating than the ActionEvent class. For our DragSmiley program, we use the MouseEvent class, which requires importing the javafx.scene.input package. We're able to take advantage of mouse events by registering the face ImageView object with three mouse events—pressing down, releasing, and dragging. Specifically, we use face to call these three setOn mouse event methods:



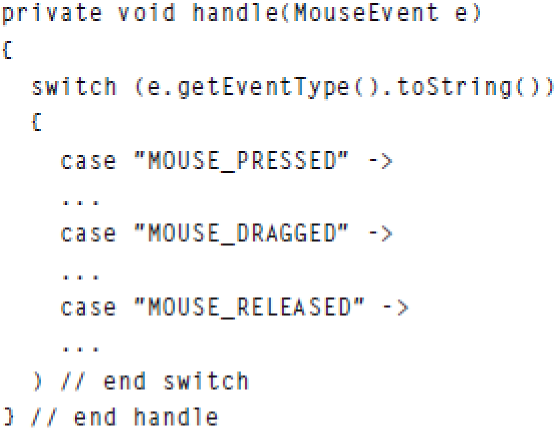
```
  public void start(Stage stage)
  {
    Pane pane = new Pane();
    Scene scene = new Scene(pane, WIDTH, HEIGHT);

    pane.getChildren().add(face);
    face.setImage(SMILEY);
    face.setOnMousePressed(this::handle);
    face.setOnMouseDragged(this::handle);
    face.setOnMouseReleased(this::handle);
    stage.setTitle("Drag Smiley");
    stage.setScene(scene);
    stage.show();
  } // end start

  //*******************************************************

  private void handle(MouseEvent e)
  {
    switch (e.getEventType().toString())
    {
      case "MOUSE_PRESSED" ->
      {
        face.setImage(SCARED);
        oldMouseX = e.getX();
        oldMouseY = e.getY();
      }
      case "MOUSE_DRAGGED" ->
      {
        double deltaX = e.getX() - oldMouseX;
        double deltaY = e.getY() - oldMouseY;
        face.setX(face.getX() + deltaX);
        face.setY(face.getY() + deltaY);
        oldMouseX += deltaX;
        oldMouseY += deltaY;
      }
      case "MOUSE_RELEASED" -> face.setImage(SMILEY);
    } // end switch
  } // end handle method
} // end class DragSmiley
```

Figure 19.17b DragSmiley program—part B

```
face.setOnMousePressed(this::handle);
face.setOnMouseReleased(this::handle);
face.setOnMouseDragged(this::handle);
```

Note that all three method calls use the same method-reference argument, this::handle, thus relying on a single event handler for all three mouse events.

Because of the face object’s registration code shown above, the handle event handler automatically receives a MouseEvent object when the mouse is on the face object and the user presses the mouse button, releases the mouse button, or drags the mouse with the button depressed. In the event handler, note the MouseEvent parameter e:



```
private void handle(MouseEvent e)
{
  switch (e.getEventType().toString())
  {
    case "MOUSE_PRESSED" ->
    ...
    case "MOUSE_DRAGGED" ->
    ...
    case "MOUSE_RELEASED" ->
    ...
  } // end switch
} // end handle
```

Note how we use e to call the getEventType and toString methods and then compare the result to the three event types.

Let’s now consider the dragging process. When the user drags the image, we need a way to keep track of the drag path. We do this by saving and updating the current mouse position in the scene. The first time we save this position is when the user presses the mouse button down. Then whenever a mouse-drag operation fires a mouse event, we compute the change in mouse position from the previous mouse position. Then we move the image and correct the saved mouse position by the amount of the change.

Let’s zoom in and examine the code for each of the three cases. Here’s the code for the mouse pressed event:

```
case "MOUSE_PRESSED" ->
{
  face.setImage(SCARED);
  oldMouseX = e.getX();
  oldMouseY = e.getY();
}
```

The oldMouseX and oldMouseY instance variables are the saved horizontal and vertical positions of the mouse. Whenever the user presses the mouse while the cursor is within face’s scope, the code changes the image to the scared one and sets the oldMouseX and oldMouseY variables to the current mouse position relative to the upper left corner of the event source, which is the face image.

Here’s the code for the mouse dragged event:

```
case "MOUSE_DRAGGED" ->
{
  double deltaX = e.getX() - oldMouseX;
  double deltaY = e.getY() - oldMouseY;
  face.setX(face.getX() + deltaX);
  face.setY(face.getY() + deltaY);
  oldMouseX += deltaX;
  oldMouseY += deltaY;
}
```

As the user drags the mouse, the program captures the change in mouse position and changes the face position by an equal amount.

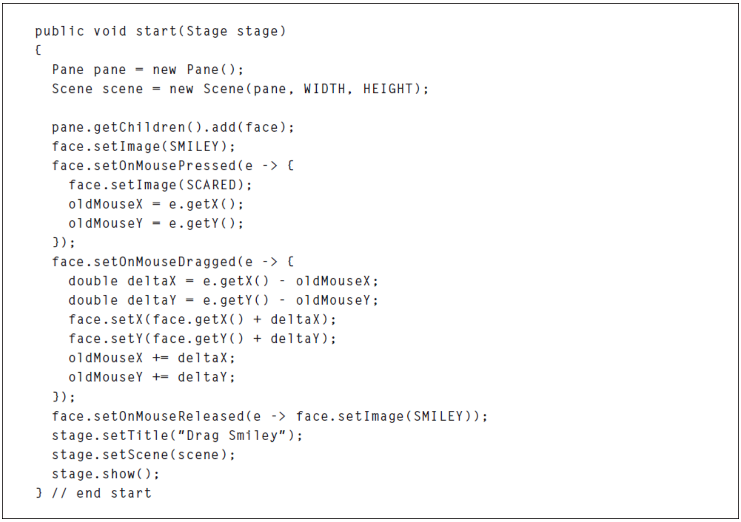
Here's the code for the mouse released event:

```
case "MOUSE_RELEASED" -> face.setImage(SMILEY);
```

When the user releases the mouse, the program resets the image to the smiley face.

DragSmiley Event Handling with Lambda Expressions

As an alternative to using method references as in the previous subsection, we can shorten the DragSmiley program by using lambda expressions. See Figure 19.18. It shows a modified start method; this time, the three setOn method calls include lambda expressions for their arguments instead of method references. With no method references, there's no need for a separate event-handler method.



```
public void start(Stage stage)
{
  Pane pane = new Pane();
  Scene scene = new Scene(pane, WIDTH, HEIGHT);

  pane.getChildren().add(face);
  face.setImage(SMILEY);
  face.setOnMousePressed(e -> {
    face.setImage(SCARED);
    oldMouseX = e.getX();
    oldMouseY = e.getY();
  });
  face.setOnMouseDragged(e -> {
    double deltaX = e.getX() - oldMouseX;
    double deltaY = e.getY() - oldMouseY;
    face.setX(face.getX() + deltaX);
    face.setY(face.getY() + deltaY);
    oldMouseX += deltaX;
    oldMouseY += deltaY;
  });
  face.setOnMouseReleased(e -> face.setImage(SMILEY));
  stage.setTitle(“Drag Smiley”);
  stage.setScene(scene);
  stage.show();
} // end start
```

Figure 19.18 DragSmiley program start method using lambda expressions

With the lambda-expression solution, in addition to eliminating the event-handler method, we also eliminate the need for the javafx.scene.input.MouseEvent import statement. Note the third lambda expression. Because it has only one statement, we omit the curly braces and that one statement’s final semicolon, reducing the entire MouseReleased event-handling method to just this:

```
    e -> face.setImage(SMILEY)
```

How’s that for compact elegance!

19.9 Lunar Eclipse Program with Circle, RadialGradient, and Slider

In the prior section, we implemented a program that enables a user to drag a smiley image with a mouse. Wasn’t that fun? In this section, we implement a program that enables a user to drag a circle shape with a slider control. Even more fun! See Figure 19.19. It shows a sequence of outputs from a LunarEclipse program that simulates an eclipse of the moon by the earth. In the first window, the earth’s shadow is at the left and does not overlap the moon. You can’t see the shadow because it displays with the same color as the background. In the second window, the earth's shadow has been moved to the right so it partially covers the moon. In the third and fourth windows, the earth's shadow completely covers the moon and then partially covers the right side of the moon.

In the program, the root of the scene graph is a BorderPane. The BorderPane's bottom area gets a Slider control. Its center area gets a Pane container. You might recall from the Pane section at the end of Chapter 18 that the Pane class doesn't provide automatic layout for its components. So in our LunarEclipse program, we explicitly position the Pane container's two components—a Circle shape for the moon and another Circle shape for the earth's shadow.

Circle, RadialGradient, and Slider Classes

In Figure 19.19, note that the moon is white at its center and transitions gradually to dark gray at its perimeter. We implement that effect by applying a RadialGradient object to the moon's Circle shape. As you'll see when we examine the code, the RadialGradient object specifies three colors at three "stop" positions along a line from the circle's center to its perimeter. At the 30% position, we specify white, at the 70% position, we specify light gray, and at the perimeter, we specify dark gray. We use this color gradation scheme to make the moon look spherical.

Note the slider at the bottom of each window. A slider allows the user to select a value from a range of values. To select a value, the user drags the slider's *thumb* along a bar. Although we don't do it here, it's possible to associate tick marks with that bar. Positions along the bar correspond to a range of values. In the LunarEclipse example, those values are pixel positions along the x axis for the center of the circle that forms the earth's shadow. As the user drags the slider's thumb to the right, the earth's shadow moves to the right. As you'll see when we examine the code, we make that connection by registering an event handler with the slider's setOnMouseDragged method.

LunarEclipse Style Sheet

The LunarEclipse program uses the style sheet shown in Figure 19.20. It prescribes the window's dimensions and background color. The .earth rule is for the earth's shadow, and the shadow is normally the same color, midnightblue, as the window's background. As explained earlier, the moon displays with a gradient that transitions from white at its center to dark gray at its perimeter. We implement that fancy formatting with method calls in the Java source code, not with JavaFX CSS rules.

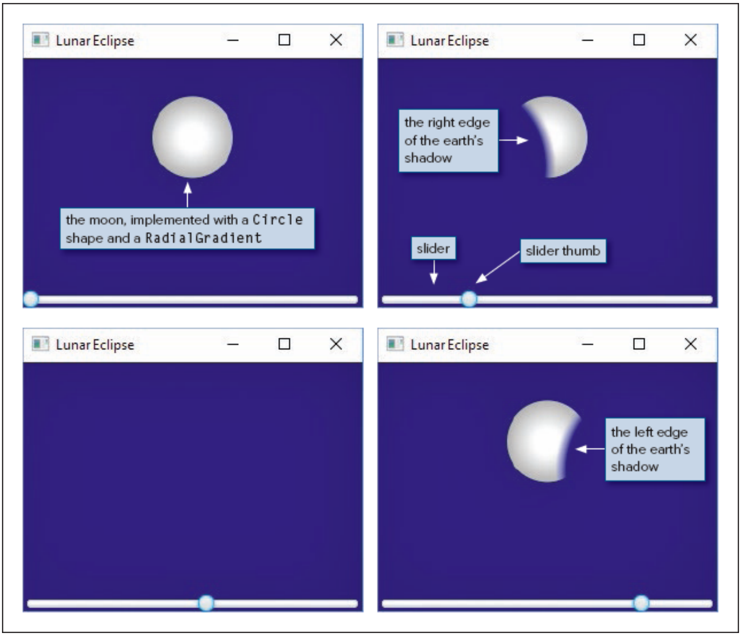

Lunar Eclipse
the moon, implemented with a Circle shape and a RadialGradient
Lunar Eclipse
the right edge of the earth's shadow
slider
slider thumb
Lunar Eclipse
Lunar Eclipse
the left edge of the earth's shadow

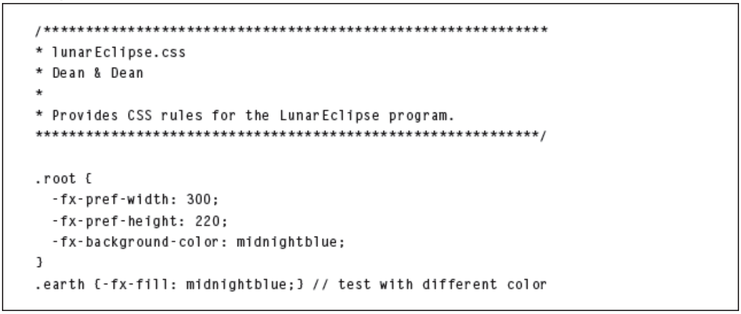
Figure 19.19 Using the LunarEclipse program's slider to show different eclipse positions

©JavaFX

In the style sheet, note the comment at the right of the .earth rule. As part of the program's development, it's easier to understand what's going on if the earth's shadow is visible. To make the shadow visible, simply edit the .earth rule by replacing midnightblue with a different color. Figure 19.21 shows the resulting window when #3040A0 is used for the earth's shadow.

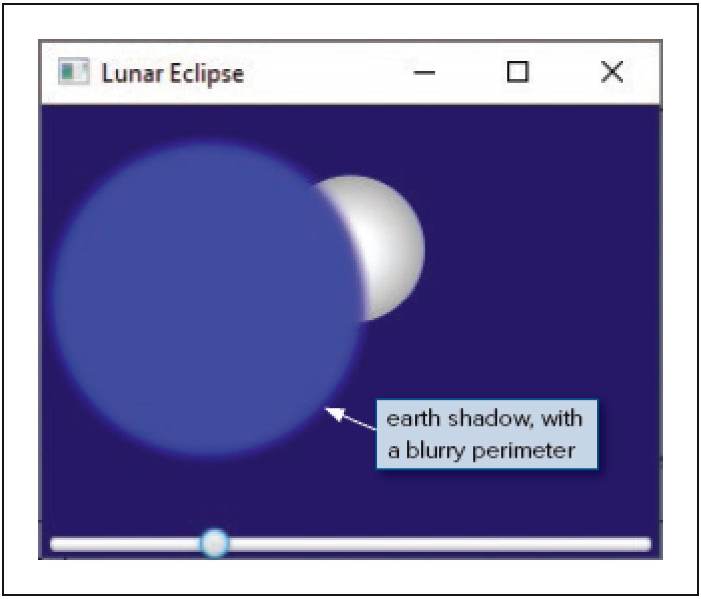
LunarEclipse Program Walk-Through

Figure 19.22a shows the first part of the LunarEclipse program. The start method is pretty simple and its statements should look familiar by now. But there is a new wrinkle: The createContents method call comes after the stage.show method call. This enables program code in the createContents method to retrieve style sheet values (like the -fx-pref-width value from the .root rule) that have been applied as part of the show method's execution. For example, in the createContents method, we use the following code to specify the moon's width as 12% of the width of the program's window:



```
/*******************************************************************
* lunarEclipse.css
* Dean & Dean
*
* Provides CSS rules for the LunarEclipse program.
*******************************************************************/

.root {
  -fx-pref-width: 300;
  -fx-pref-height: 220;
  -fx-background-color: midnightblue;
}
.earth {-fx-fill: midnightblue;} // test with different color
```

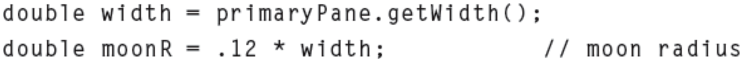
Figure 19.20 Style sheet for LunarEclipse program



earth shadow, with a blurry perimeter

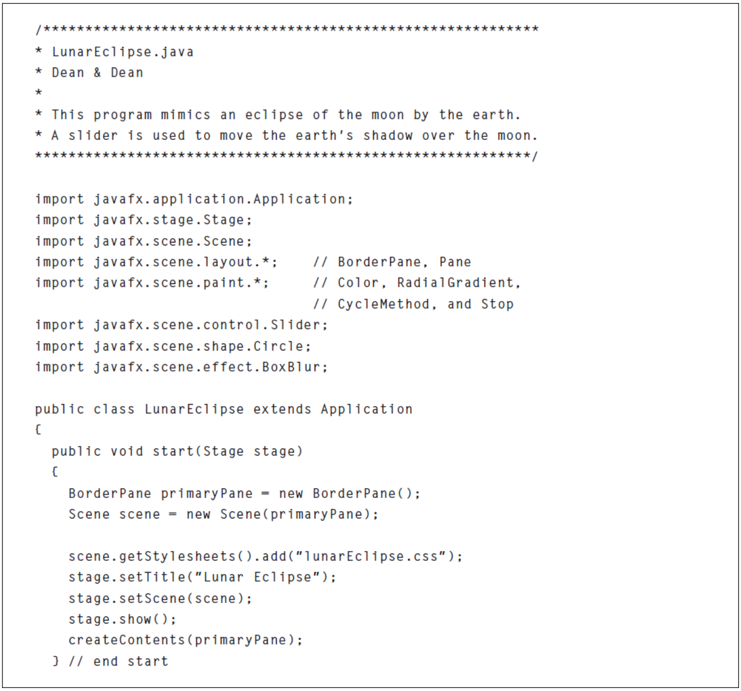
Figure 19.21 LunarEclipse program's display when the earth's shadow uses a different color

©JavaFX



```
double width = primaryPane.getWidth();
double moonR = .12 * width;          // moon radius
```

Figure 19.22b shows the createContents method. We first retrieve the displayed window’s width and height, and we then use those values as part of the initialization statements for the moon’s radius and -center point position and the earth shadow’s radius and center point position. By using the window’s dimensions for the shapes, the shapes become larger or smaller if we update the program later on with a larger or smaller window.

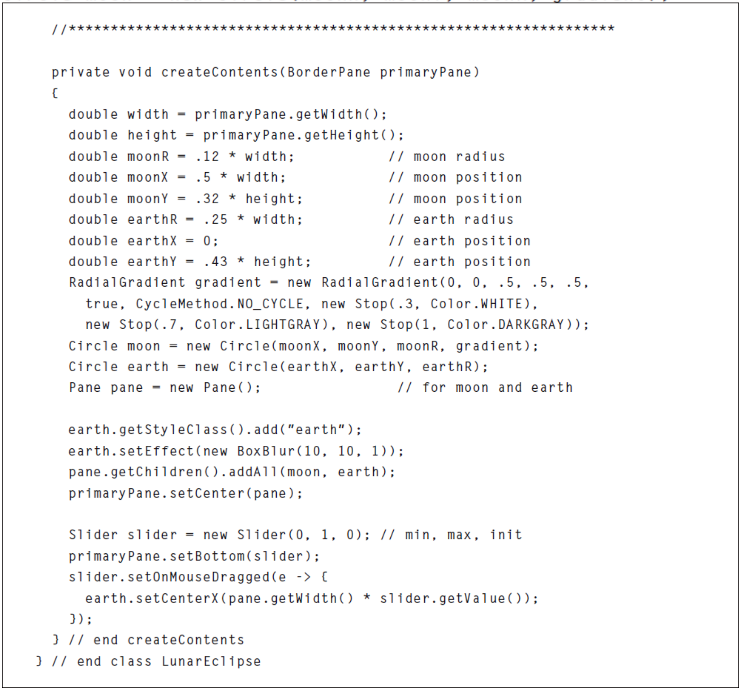


```
/************************************************************
* LunarEclipse.java
* Dean & Dean
*
* This program mimics an eclipse of the moon by the earth.
* A slider is used to move the earth's shadow over the moon.
************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;    // BorderPane, Pane
import javafx.scene.paint.*;     // Color, RadialGradient,
                                 // CycleMethod, and Stop
import javafx.scene.control.Slider;
import javafx.scene.shape.Circle;
import javafx.scene.effect.BoxBlur;

public class LunarEclipse extends Application
{
  public void start(Stage stage)
  {
    BorderPane primaryPane = new BorderPane();
    Scene scene = new Scene(primaryPane);

    scene.getStylesheets().add("lunarEclipse.css");
    stage.setTitle("Lunar Eclipse");
    stage.setScene(scene);
    stage.show();
    createContents(primaryPane);
  } // end start
```

Figure 19.22a LunarEclipse program—part A

Note how we call the Circle constructor to implement the moon and the earth's shadow. The Circle constructor's first three arguments are for the circle's x and y center point values and for the circle's radius. The moon's Circle constructor uses a fourth argument for a RadialGradient object. As explained earlier, the moon uses a gradient to display white at its center and transition gradually to dark gray at its perimeter. Here’s the relevant code:

```
RadialGradient gradient = new RadialGradient(0, 0, .5, .5, .5,
  true, CycleMethod.NO_CYCLE, new Stop(.3, Color.WHITE),
  new Stop(.7, Color.LIGHTGRAY), new Stop(1, Color.DARKGRAY));
Circle moon = new Circle(moonX, moonY, moonR, gradient);
```



```
  //*******************************************************************

  private void createContents(BorderPane primaryPane)
  {
    double width = primaryPane.getWidth();
    double height = primaryPane.getHeight();
    double moonR = .12 * width;           // moon radius
    double moonX = .5 * width;            // moon position
    double moonY = .32 * height;          // moon position
    double earthR = .25 * width;          // earth radius
    double earthX = 0;                    // earth position
    double earthY = .43 * height;         // earth position
    RadialGradient gradient = new RadialGradient(0, 0, .5, .5, .5,
      true, CycleMethod.NO_CYCLE, new Stop(.3, Color.WHITE),
      new Stop(.7, Color.LIGHTGRAY), new Stop(1, Color.DARKGRAY));
    Circle moon = new Circle(moonX, moonY, moonR, gradient);
    Circle earth = new Circle(earthX, earthY, earthR);
    Pane pane = new Pane();                // for moon and earth

    earth.getStyleClass().add("earth");
    earth.setEffect(new BoxBlur(10, 10, 1));
    pane.getChildren().addAll(moon, earth);
    primaryPane.setCenter(pane);

    Slider slider = new Slider(0, 1, 0); // min, max, init
    primaryPane.setBottom(slider);
    slider.setOnMouseDragged(e -> {
      earth.setCenterX(pane.getWidth() * slider.getValue());
    });
  } // end createContents
} // end class LunarEclipse
```

Figure 19.22b LunarEclipse program—part B

Creates a new instance of RadialGradient.

Yes, we know that the above description is somewhat lacking in terms of specificity. If you drill down on the API web page, you can get more details, but they're pretty confusing. Bottom line: Use the first seven arguments as shown above (0, 0, .5, .5, .5, true, CycleMethod.NO_CYCLE) for normal gradients where the gradient's center point matches the center point of the circle that the gradient is applied to and the gradient's specified colors start at the center and end at the circle's perimeter.

Continuing with our examination of the RadialGradient constructor's arguments, the subsequent arguments are Stop objects, each of which specifies a color and a position where that color appears along the radial line that extends out from the gradient's center. For example, here's the first Stop object:

```
new Stop(.3, Color.WHITE)
```

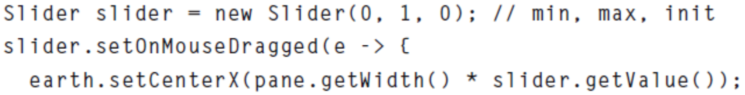
That Stop object means that white starts at the center point and extends 30% of the way to the perimeter of the gradient's surrounding Circle object. From 30% to 70%, the color transitions from white to light gray (the color at the second Stop object). From 70% to the circle's perimeter, it transitions from light gray to dark gray.

Glance back at Figure 19.21 and note the earth shadow's blurry edges. That blurriness occurs in the real world due to the sun not being a point source. We use a BoxBlur object to engineer that special effect. Here's the relevant code:

```
earth.setEffect(new BoxBlur(10, 10, 1));
```

The BoxBlur class uses an image filtering technique such that each of the image's pixels is converted to a new color which equals the average color value of its neighboring pixels. The constructor's first two arguments specify the width and height of the rectangle of pixels (the “box”) in which the averaging takes place. The third argument specifies the number of times the blurring process is performed on the image.

Next up, let’s examine the code for the Slider control. Specifically, let’s examine these statements at the bottom of the createContents method:



```
Slider slider = new Slider(0, 1, 0); // min, max, init
slider.setOnMouseDragged(e -> {
  earth.setCenterX(pane.getWidth() * slider.getValue());
```

The slider.getValue() method call retrieves the slider’s value, which is a number commensurate with the current position of the slider’s thumb. The Slider constructor’s first two arguments specify the slider’s minimum and maximum values. So with argument values of 0 and 1 in the above constructor call, if the thumb is a quarter of the way from the left, the slider’s value would be .25. The constructor’s third argument specifies the thumb’s initial position.

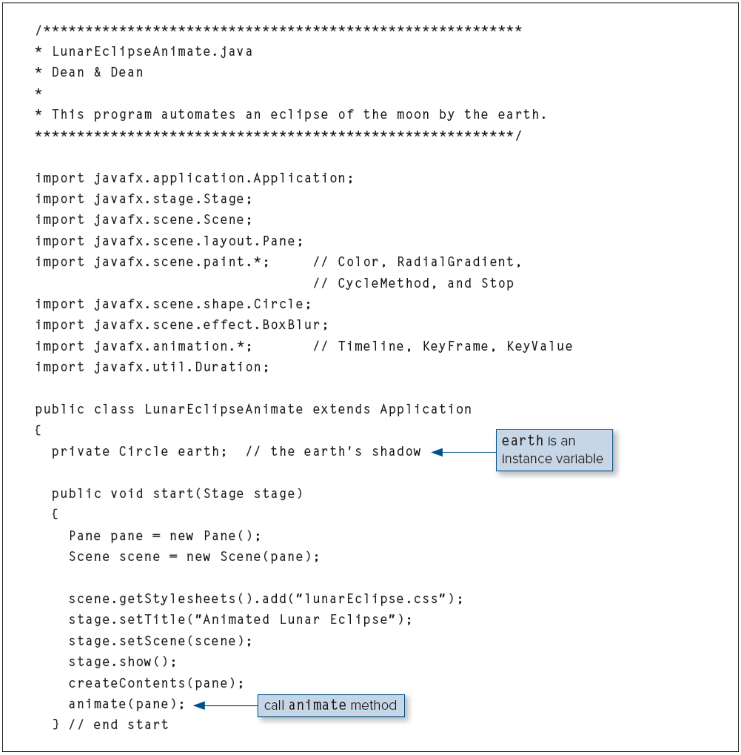
In the slider control code, note how we register an event handler with the slider for when the mouse drags the slider's thumb. We call the setOnMouseDragged method and use a lambda expression for the method call's argument. The lambda expression retrieves the slider's value (a number between 0 and 1), multiplies it by the width of the pane used to display the eclipse, and uses the product as the argument in earth's call to setCenterX. The end result is that the earth's shadow gets positioned so it aligns with the slider's thumb.

19.10 Animation

In the last two sections' programs, the user was involved in making something move—the user moved the smiley face image by dragging the image and the user moved the earth's shadow by dragging the slider thumb.

In this section, we modify the lunar eclipse program by replacing the slider with animation, where the earth's shadow moves left to right over the moon on its own accord.

Our new LunarEclipseAnimate program is very similar to the original LunarEclipse program. The only differences are that there's no slider, earth is an instance variable instead of a local variable, and there's an animate helper method that performs the animation. See Figure 19.23a, which shows the first part of the LunarEclipseAnimate program. As promised, earth (the Circle shape for the earth's shadow) is now an instance variable. We use an instance variable to facilitate sharing between the createContents and animate methods. Note that at the bottom of the start method, after calling createContents, we call animate, which performs the animation on the earth shape.



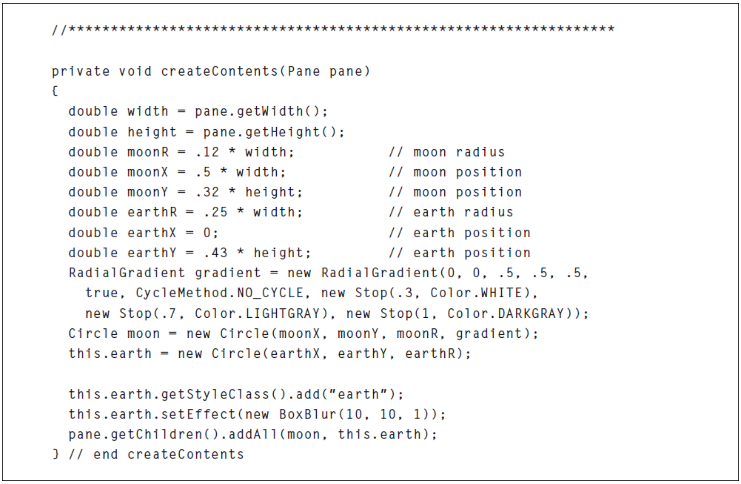
```
/*************************************************************
* LunarEclipseAnimate.java
* Dean & Dean
*
* This program automates an eclipse of the moon by the earth.
*************************************************************/

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.*;      // Color, RadialGradient,
                                  // CycleMethod, and Stop
import javafx.scene.shape.Circle;
import javafx.scene.effect.BoxBlur;
import javafx.animation.*;        // Timeline, KeyFrame, KeyValue
import javafx.util.Duration;

public class LunarEclipseAnimate extends Application
{
  private Circle earth;  // the earth's shadow

  public void start(Stage stage)
  {
    Pane pane = new Pane();
    Scene scene = new Scene(pane);

    scene.getStylesheets().add(”lunarEclipse.css”);
    stage.setTitle(”Animated Lunar Eclipse”);
    stage.setScene(scene);
    stage.show();
    createContents(pane);
    animate(pane);
  } // end start
```

`earth` is an instance variable

call `animate` method

Figure 19.23a LunarEclipseAnimate program—part A

Figure 19.23b shows the createContents method, which implements all the shapes. It's the same as in the original program, except that there's no slider code.



```
//***********************************************************************

private void createContents(Pane pane)
{
  double width = pane.getWidth();
  double height = pane.getHeight();
  double moonR = .12 * width;          // moon radius
  double moonX = .5 * width;           // moon position
  double moonY = .32 * height;         // moon position
  double earthR = .25 * width;         // earth radius
  double earthX = 0;                   // earth position
  double earthY = .43 * height;        // earth position
  RadialGradient gradient = new RadialGradient(0, 0, .5, .5, .5,
    true, CycleMethod.NO_CYCLE, new Stop(.3, Color.WHITE),
    new Stop(.7, Color.LIGHTGRAY), new Stop(1, Color.DARKGRAY));
  Circle moon = new Circle(moonX, moonY, moonR, gradient);
  this.earth = new Circle(earthX, earthY, earthR);

  this.earth.getStyleClass().add("earth");
  this.earth.setEffect(new BoxBlur(10, 10, 1));
  pane.getChildren().addAll(moon, this.earth);
} // end createContents
```

Figure 19.23b LunarEclipseAnimate program—part B

Figure 19.23c shows the animate method. We use a Timeline object to set up and execute the animation. The first step is to define how long the animation will last, and we use the Duration class's seconds method to define that length in seconds. Here's how we set the duration to 5 seconds:

```
Duration duration = Duration.seconds(5);
```

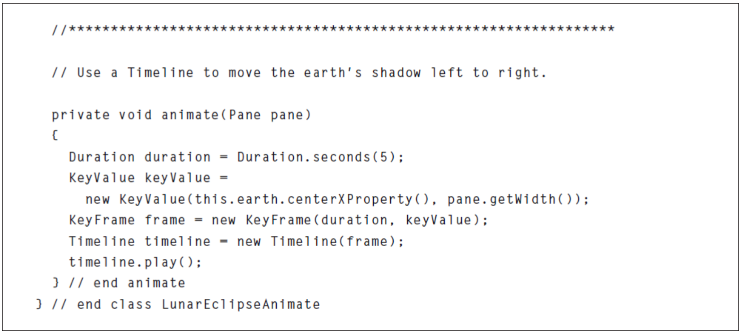
Next, we instantiate a KeyValue object to define what happens during that time interval. More specifically, we call the KeyValue constructor with two arguments—an object's property and the value that property is supposed to attain at the end of the animation. Here's the relevant code from the LunarEclipseAnimate program:

```
   KeyValue keyValue =
   new
KeyValue(this.earth.centerXProperty(),
pane.getWidth());
```

We want to move the earth shadow's circle along the x axis, so for the constructor call's first argument, we call the circle's centerXProperty method to retrieve the property that holds the x position of the circle's center point. When the program loads, the earth shadow's circle is positioned at x = 0 (x and y values refer to the number of pixels away from the enclosing container's top-left corner, where the top-left corner is at position x = 0, y = 0). For the constructor call's second argument, we call pane's getWidth method (pane is the enclosing container) to retrieve the x value of pane's right edge.

Next we bundle the duration and keyValue variables into a KeyFrame object, like this:

KeyFrame frame = new KeyFrame(duration, keyValue);



```
  //*****************************************************************

  // Use a Timeline to move the earth's shadow left to right.

  private void animate(Pane pane)
  {
    Duration duration = Duration.seconds(5);
    KeyValue keyValue =
      new KeyValue(this.earth.centerXProperty(), pane.getWidth());
    KeyFrame frame = new KeyFrame(duration, keyValue);
    Timeline timeline = new Timeline(frame);
    timeline.play();
  } // end animate
} // end class LunarEclipseAnimate
```

Figure 19.23c LunarEclipseAnimate program—part C

We then instantiate a TimeLine object with a constructor call that uses the frame variable as its argument and call the Timeline class's play method to execute the Timeline object's animation.

If you need more than one frame in your timeline animation, no problem. You can call getKeyFrames to retrieve the Timeline's set of frames, and then call setAll to add multiple frames. Here's what we're talking about:

```
timeline.getKeyFrames().setAll(frame1, frame2);
```

In addition to the play method, the Timeline class also has pause and stop methods. The pause method stops the animation, but retains the position where the animation stopped, so when play executes, it starts at the paused position. The stop method resets the animation's positon to the start. If you're looking for practice, add play, pause, and stop buttons to the LunarEclipseAnimate program.

Summary

- A TextArea control displays a multiple-line box for user input.
- A CheckBox control displays a small square with an identifying label. Users click the check box to toggle it between selected and unselected.
- A RadioButton control displays a small circle with a label to its right. If an unselected button is clicked, the clicked button becomes selected, and assuming the radio button is in a ToggleGroup, the previously selected button in the group becomes unselected.
- A ComboBox control allows the user to select an item from a list of items. ComboBox components are called “combo boxes” because they are a combination of a text box (normally, they look just like a text box) and a list (when the

down arrow is clicked, they look like a list).

- A MenuBar contains Menus, which in turn contain MenuItems.
- To display an image in a JavaFX program, store an Image object in an ImageView object and then assign the ImageView object to the scene graph.
- To handle mouse events, register with a setOn method call that corresponds to the particular event of interest.
- A Slider control enables a user to enter a variable value by dragging a slider thumb along a bar.
- To implement animation, you can use the Timeline class to transition an object’s property value to a target value for a specified duration.

Review Questions

§19.2 User Interface Design

1. Provide three user interface design guidelines for GUI programs.

§19.3 TextArea Control

2. TextArea components are editable by default. (T / F)

3. TextArea components employ line wrap by default. (T / F)

§19.4 CheckBox and RadioButton Controls

4. What happens if you click a check box that's already selected?

5. Write a code fragment that creates a check box named attendance with an "I will attend" label and then puts it into its selected state.

6. What happens if you click a radio button that is already selected?

7. What happens if you click an initially unselected radio button that is a member of a ToggleGroup?

§19.5 ComboBox Control

8. How are combo boxes and radio button groups similar?

9. What method call retrieves the current selection in a combo box?

§19.6 Job Application Program

10. In the JobApplication program, what happens if the following code is omitted?

```
ToggleGroup radioGroup = new ToggleGroup();
...
good.setToggleGroup(radioGroup);
bad.setToggleGroup(radioGroup);
```

§19.7 ScrollPane and Menu Classes

11. How are MenuBar, Menu, and MenuItem objects related?

§19.8 Images and Mouse Events

12. Can an Image object be a node in a scene graph?

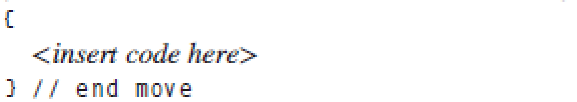
13. Provide a statement that registers a lambda expression that assigns an Image object named SCARED to an ImageView object named face when the user clicks on face.

§19.9 Lunar Eclipse Program with Circle, RadialGradient, and Slider

14. Provide a Slider constructor call with minimum value of 0, maximum value of 50, and initial value of 10.

§19.10 Animation

15. In the following code fragment, replace <*insert code here*> with code that implements an animation that moves the passed-in circle parameter from the left side of its pane container to the right side of that container over a time period of 10 seconds.



```
private void move(Circle circle, Pane pane)
{
  <insert code here>
} // end move
```

Exercises

---

1. [after §19.3] Suppose you're given a window with two TextArea components, named msg1 and msg2, and a Button component. When clicked, the button swaps the contents of the two text areas. Provide the code that performs the swap operation.

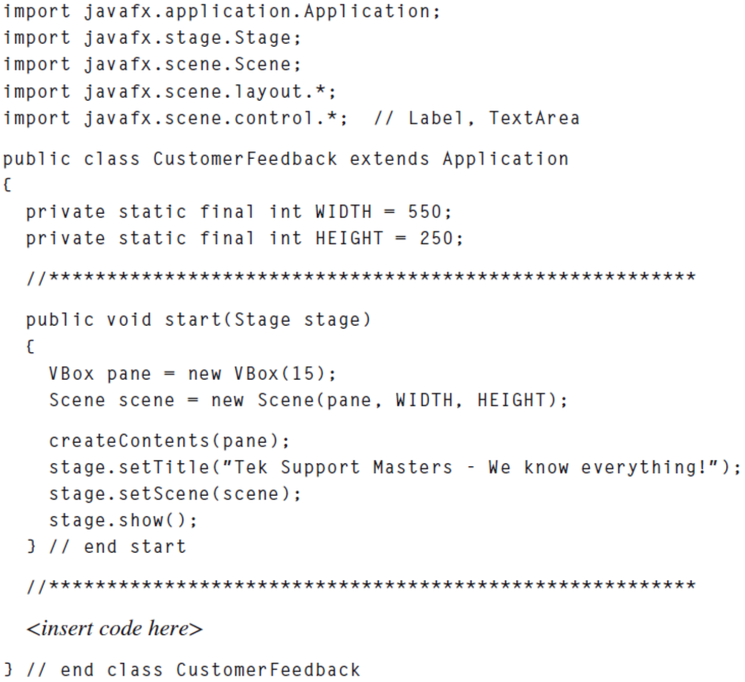
That is, provide the code that goes in the body of the button's setOnAction lambda expression or in the body of a referenced event-handling method.

2. [after §19.3] Given the following program skeleton. Replace <*insert code here*> with a createContents method such that the resulting window matches the customer feedback window shown earlier in this chapter, in the TextArea Control section. You'll want to instantiate two Label controls and a TextArea control and add them to the VBox pane. Run your completed program to make sure your generated form looks the same as what's in the book. In particular, pay attention to these things:

- The pane should have 15 pixels of padding.
- Normally, a TextArea box will

span the entire width of a VBox. Prevent that default behavior.

- User input should wrap to the next line when it bumps up against the box's right edge.



```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.*;
import javafx.scene.control.*;  // Label, TextArea

public class CustomerFeedback extends Application
{
  private static final int WIDTH = 550;
  private static final int HEIGHT = 250;

  //*******************************************************

  public void start(Stage stage)
  {
    VBox pane = new VBox(15);
    Scene scene = new Scene(pane, WIDTH, HEIGHT);

    createContents(pane);
    stage.setTitle("Tek Support Masters - We know everything!");
    stage.setScene(scene);
    stage.show();
  } // end start

  //*******************************************************

  <insert code here>

} // end class CustomerFeedback
```

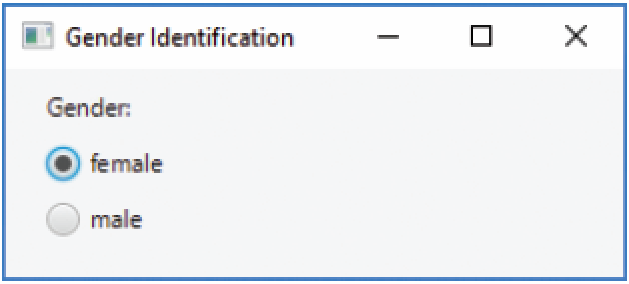
3. [after §19.4] Write a statement that creates a check box named bold. The check box should be unselected, and it should have a label that says "boldface type".

4. [after §19.4] How can your code determine whether a check box is selected or not?

5. [after §19.4] Given a gender.css file with the following contents:

```
.root {
  -fx-padding: 10 20 20 20;
  -fx-min-width: 300;
}
```

And the following desired initial display:



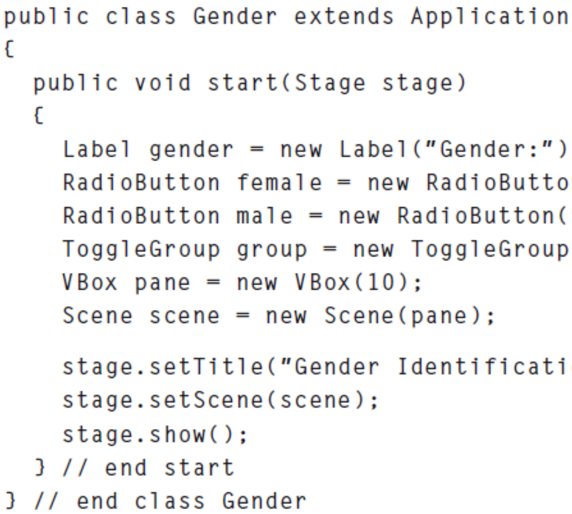
Gender Identification

Gender:

● female

○ male

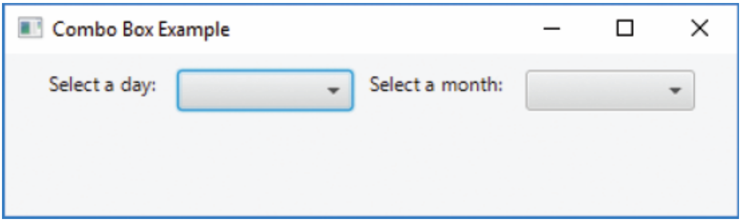
Finish the skeleton program below so that the female and male radio buttons behave in the normal fashion—when one is selected, the other is unselected. Note that the female button is selected when the window initially displays. Here is starting skeleton:



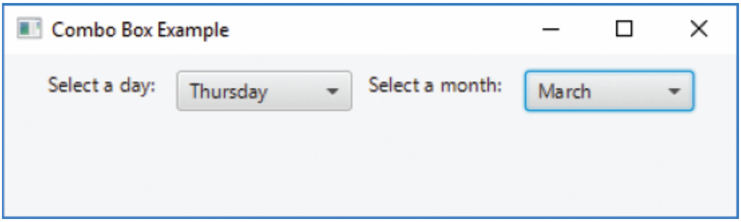
```
public class Gender extends Application
{
  public void start(Stage stage)
  {
    Label gender = new Label(”Gender:”);
    RadioButton female = new RadioButton(”female”);
    RadioButton male = new RadioButton(”male”);
    ToggleGroup group = new ToggleGroup();
    VBox pane = new VBox(10);
    Scene scene = new Scene(pane);

    stage.setTitle(”Gender Identification”);
    stage.setScene(scene);
    stage.show();
  } // end start
} // end class Gender
```

6. [after §19.5] The CheckBox, RadioButton, and ComboBox components are defined in what package(s)?

7. [after §19.6] Provide a createContents method and a style sheet for a program that initially displays this window:



After the user clicks the Select a day down arrow and selects Thursday and clicks the Select a month down arrow and selects March, the program’s display looks like this:



Select a day: Thursday

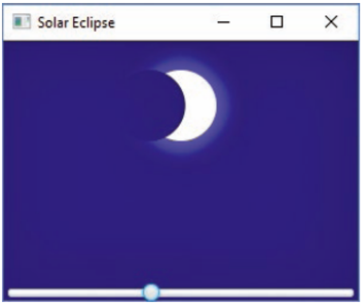
Select a month: March

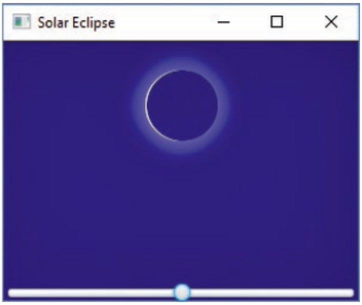
Your createContents method must work in conjunction with this program skeleton:

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.HBox;
import javafx.scene.control.*;    // Label, ComboBox

public class ComboBoxExample extends Application
{
  private ComboBox<String> daysBox = new ComboBox<>();
  private ComboBox<String> monthsBox = new ComboBox<>();
  private String[] days =
    {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
  private String[] months = {"January", "February", "March",
    "April", "May", "June", "July", "August", "September",
    "October", "November", "December"};

  public void start(Stage stage)
  {
    HBox pane = new HBox(10);
    Scene scene = new Scene(pane);

    scene.getStylesheets().add("comboBox.css");
    createContents(pane);
    stage.setTitle("Combo Box Example");
    stage.setScene(scene);
    stage.show();
  } // end start

  <insert createContents method here>

} // end class ComboBoxExample
```

8. [after §19.8] Using JavaFX's API website, look up the MouseEvent class and list all of its static constant EventType values.

9. [after §19.9] Modify the text's slider-driven LunarEclipse program to make it simulate a solar eclipse. The main change will be adding a corona shape on top of the sun shape. The sun's corona is the atmosphere surrounding the sun's surface. Normally, one cannot view it (or the sun) directly. But during a solar eclipse, it becomes more apparent. Here's what your resulting window should look like:





Make the sun's radius equal to 0.1

times the primary pane's width, and put it where the LunarEclipse program's moon was. Make the new program's moon have the same diameter as its sun and make this moon's vertical position match the sun's vertical position. Use a slider to control the horizontal position of the new program's moon. Use the following constructors for the sun, the sun's corona, and the moon:

```
Circle sun = new Circle(sunX, sunY, sunR, Color.WHITE);
Circle corona = new Circle(
  sunX, sunY, 1.3 * sunR, new Color(1.0, 1.0, 1.0, 0.3));
Circle moon = new Circle(
  moonX, moonY, moonR, new Color(0.16, 0.16, 0.4, 1.0));
```

And use this method call to blur the sun's corona:

```
corona.setEffect(new BoxBlur(sunR, sunR, 1));
```

10. [after §19.10] Implement a program that animates the solar eclipse described in the previous exercise. Omit the previous exercise's slider.

Review Question Solutions

1. Three user interface design guidelines for GUI programs:

- Choose the right components.
- Position components appropriately.
- Be consistent.

2. True. TextArea components are editable by default.

3. False. TextArea components do not employ line wrap by default.

4. If you click a check box that's already selected, the check box becomes unselected.

5. The following code creates a check box named attendance with an "I will attend" label and then puts it into its selected state:

```
CheckBox attendance = new CheckBox("I will attend");
attendance.setSelected(true);
```

6. Nothing. It stays selected.

7. The clicked button becomes selected and all other buttons in the group become unselected.

8. Combo boxes and radio button groups are similar in that they both allow the user to select one item from a list of items.

9. To retrieve the current selection in a combo box, call getValue.

10. If the radioGroup code is omitted from the JobApplication program, the program still compiles and runs, but the radio buttons operate independently. In other words, clicking one radio button will not cause the other one to be unselected.

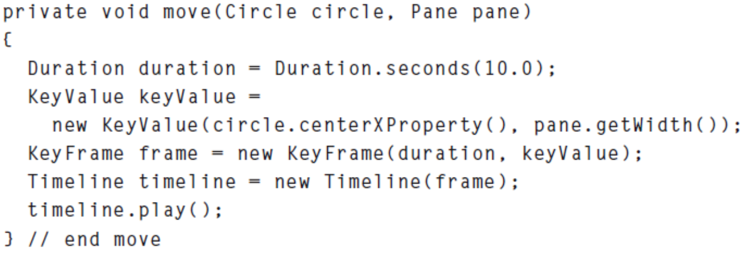
11. A MenuBar object contains Menu objects, and Menu objects contain MenuItem objects.

12. No, an Image object cannot be a node in a scene graph. You must embed the image in an ImageView object, and add the ImageView object to the scene graph.

13. face.setOnMousePressed(e -> face.setImage(SCARED));

14. Slider constructor call:
  new Slider(0, 50, 10);

15. Implement a move method.



```
private void move(Circle circle, Pane pane)
{
  Duration duration = Duration.seconds(10.0);
  KeyValue keyValue =
    new KeyValue(circle.centerXProperty(), pane.getWidth());
  KeyFrame frame = new KeyFrame(duration, keyValue);
  Timeline timeline = new Timeline(frame);
  timeline.play();
} // end move
```

---

[1] *gif* stands for Graphics Interchange Format. Typically, the GIF file format is used for simple drawn images. GIF files tend to be small and their file sizes can be reduced further with lossless data compression techniques. An alternative to GIF is the Joint

Photographic Experts Group (*JPEG*) format, which uses the .jpg extension. Typically, the JPEG file format is used to store more complex images, such as photographs. JPEG file sizes can be reduced with data compression techniques, but such compression can lead to information loss.

Multithreading

📖 To understand this appendix, you need to be familiar with object-oriented programming, inheritance, and exception handling. As such, you need to have read up through Chapter 15.

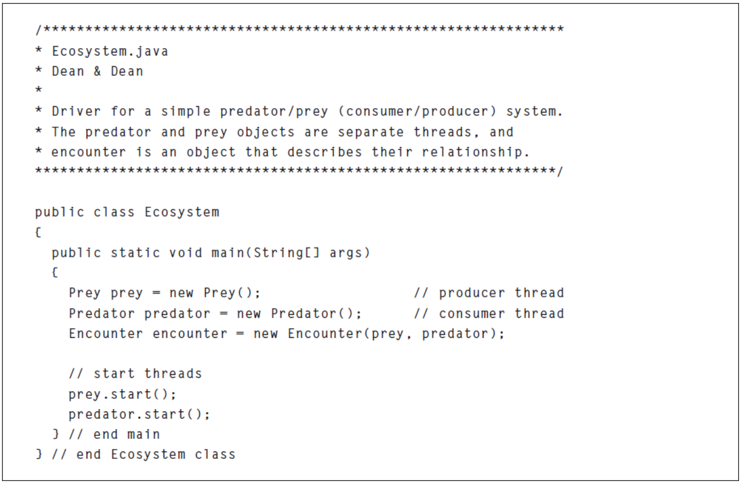
This appendix introduces a feature, *multithreading,* that helps Java programs take advantage of the parallel processing capabilities contained in many modern computers. By taking advantage of parallel processing capabilities, multithreading can lead to faster programs. And that in turn leads to more user-friendly programs.

Threads

A *thread* is a “lightweight process.” 1 Think of it as a coherent code fragment. Ideally, once it has a certain minimum amount of initial information, a thread can run all the way to completion without any more information from the outside world. Ideally, different threads are independent.

A thread is an object derived from a class that extends the predefined Thread class, and you must override the Thread class’s run method to specify what you want your thread to do. You can call an object’s run method only once, and you must do it indirectly by calling the public void start() method, which your class inherits from the Thread class. The start method asks the JVM to call your run method.2

This operation makes the newly started thread run in parallel with the software that called it. After thus starting one thread, your software could start another thread, and you would have three chunks of code running in parallel. You could continue like this to obtain as many parallel operations as you might want. Your computer automatically takes advantage of these relatively independent chunks of code to keep its various parallel hardware components as busy as possible. The driver in Figure A9.1 shows how easy it is to launch new threads.



```
/*************************************************************
* Ecosystem.java
* Dean & Dean
*
* Driver for a simple predator/prey (consumer/producer) system.
* The predator and prey objects are separate threads, and
* encounter is an object that describes their relationship.
*************************************************************/

public class Ecosystem
{
  public static void main(String[] args)
  {
    Prey prey = new Prey();                    // producer thread
    Predator predator = new Predator();        // consumer thread
    Encounter encounter = new Encounter(prey, predator);

    // start threads
    prey.start();
    predator.start();
  } // end main
} // end Ecosystem class
```

Figure A9.1 Top level of a program that simulates a simple ecosystemThis class drives the classes in Figures A9.2, A9.3, and (A9.4 or A9.5a and A9.5b).

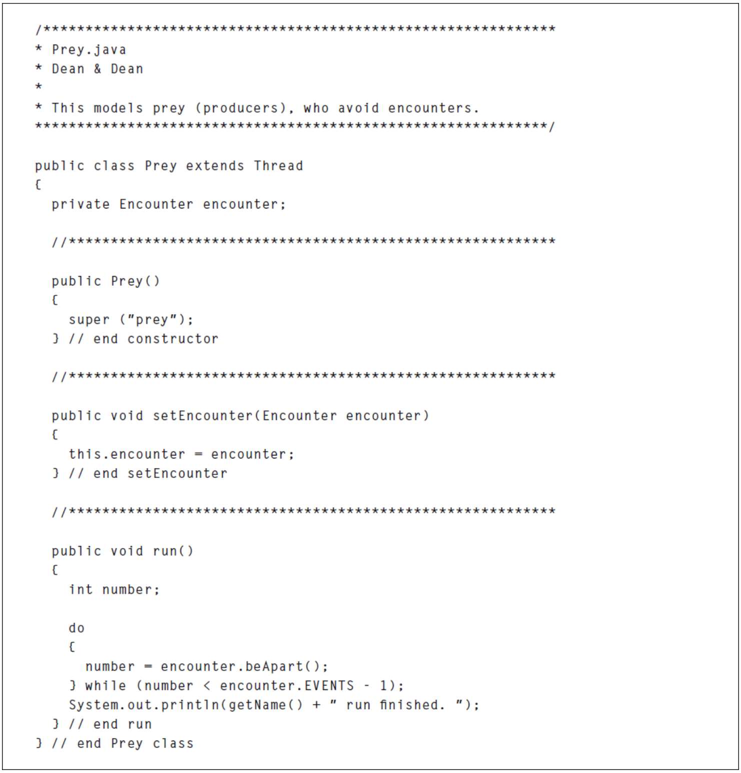
The class in Figure A9.1 is the top level of a program that describes the interaction of a predator like a fox and prey like a group of field mice. In this example, the prey gets its food continuously from ever-present vegetation, while the predator gets its food intermittently by eating prey when predator and prey happen to meet. The prey is one thread. The predator is another thread. Notice how this driver starts both prey and predator threads. These threads represent the parallel lives of these creatures in their ecosystem.

For simplicity, we restrict our discussion to classes that implement the Thread class.

The prey and predator threads are objects. There is also another object, called encounter. This object represents an ongoing intermittent relationship between the predator and the prey. In this relationship, some of the prey come into the presence of the predator, and the predator eats them. Presumably the predator eats only part of the prey in each particular encounter, and in the interim the prey continuously replenish by reproducing and eating vegetation. In the encounter relationship, the prey provides food, and the predator consumes food. So computer folks like us might say the prey thread is a *producer* thread, and the predator thread is a *consumer* thread. Of course, the prey also "consume" vegetation, so if our model included a relationship between the field mice and the vegetation they eat, in that context we could call our prey thread a "consumer" thread. So the terms "producer" and "consumer" should not be associated absolutely to any one thread.

Any relationship between threads violates the ideal of "thread independence." It complicates the lives of real creatures, and it complicates a program that simulates them.

Figure A9.2 shows the class that describes prey threads. Notice that it does extend the Thread class. There is just one instance variable, a reference to the encounter relationship—field mice are undoubtedly aware of their unpleasant relationship with a fox. The zero-parameter constructor assigns a default name to all objects of the class. That's sufficient for our example, because our driver creates only one such object, but you could also provide a one-parameter constructor to assign different names to different thread instances. The public setEncounter method allows the outside world to set the encounter reference at any time after Prey thread instantiation. The run method is the heart of a thread's definition. In this case, it's pretty simple. What prey want is to "be apart," so the run method calls the relationship's beApart method.



```
/***************************************************************
* Prey.java
* Dean & Dean
*
* This models prey (producers), who avoid encounters.
***************************************************************/

public class Prey extends Thread
{
  private Encounter encounter;

  //*************************************************************

  public Prey()
  {
    super (”prey”);
  } // end constructor

  //*************************************************************

  public void setEncounter(Encounter encounter)
  {
    this.encounter = encounter;
  } // end setEncounter

  //*************************************************************

  public void run()
  {
    int number;

    do
    {
      number = encounter.beApart();
    } while (number < encounter.EVENTS - 1);
    System.out.println(getName() + ” run finished. ”);
  } // end run
} // end Prey class
```

Figure A9.2 Class describing prey (producers) who want to escape from predators This is driven by the class in Figure A9.1.

Figure A9.3 shows the class that describes Predator threads. It also extends the Thread class. It also has an instance variable that refers to the encounter relationship—a fox is certainly aware of its pleasant relationship with field mice. It also has a zero-parameter constructor, and it also has a setEncounter method.

Notice that Predator also declares an array of delay times. With appropriate cross referencing, we could have put this program's time-delay information in any of the classes. But because the predator is the primary "cause" of encounters, we elected to put it in Predator's definition and implement it in the Predator's run method. This time-delay implementation makes Predator's run method more complicated than Prey's run method. We implement each delay by passing an integer number to the pre-written sleep method, which is in the Thread class in the always-available java.lang package:

```
public static void sleep(long millis)
    throws InterruptedException
```

What does this sleep method do? It makes the currently executing thread cease its execution for a number of milliseconds equal to the parameter value. In our example, the first element in the DELAY array is 2347, so when you run the program, you will experience a pause of 2.347 seconds between the first and second screen outputs.

Notice that the sleep method can throw an InterruptedException. If you look up InterruptedException, you'll find that it's derived directly from the Exception class, so it is a checked exception. Therefore, the method call that might throw this exception must be in a try block, and that's where we put it. Our program never does anything that might cause this exception to be thrown,[3] so we use an *empty* catch block. For better debugging feedback, you could put something like e.printStackTrace() in the catch block.

Now let's look at a first crude

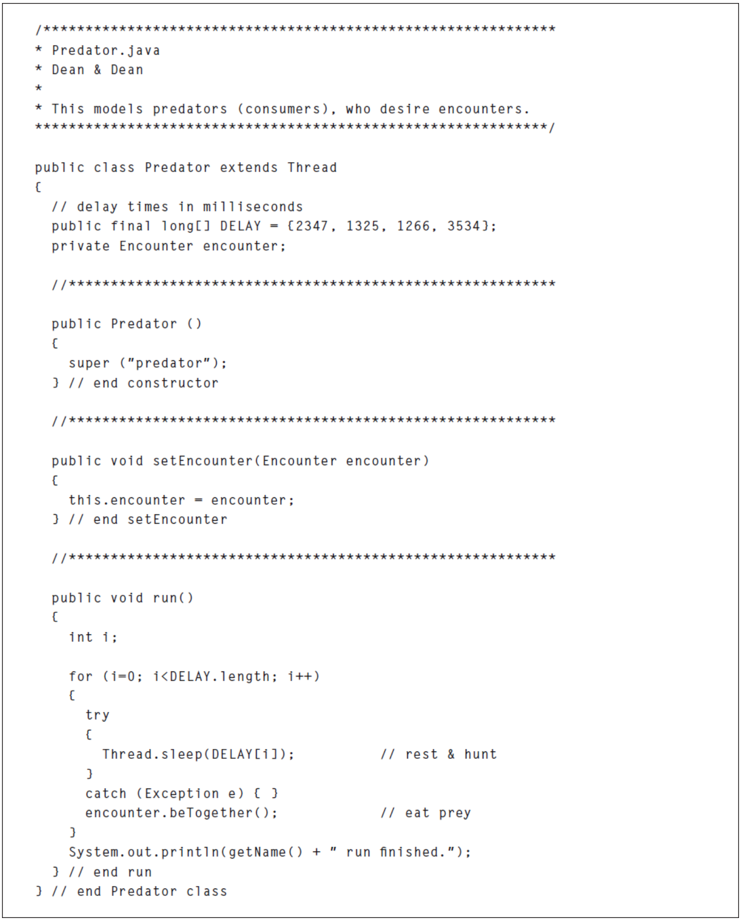
attempt to implement the Encounter class, which appears in Figure A9.4. In the lives of a single predator and a group of its prey (our chosen threads), encounters occur several times. We might have written our Encounter class so that each encounter object represented one *discrete event*, but it's easier to keep track of time and space relationships if you group related events together. Thus, one of our encounter objects represents a complete sequence of encounters between our predator thread and our prey thread. In simulation programming, this kind of on-going relationship is usually called a *process*.

The instance variables in the encounter object keep track of the total number of events, the sequence number of the current event, and references to the prey and predator threads. If you look back at Figure A9.1, you'll see that we call the

Encounter constructor after we call the Prey and Predator constructors. This calling sequence enables us to pass predator and prey references to the - encounter object when we instantiate it. Then, in the Encounter constructor we reciprocate by sending an - encounter reference to the to the predator and prey objects.

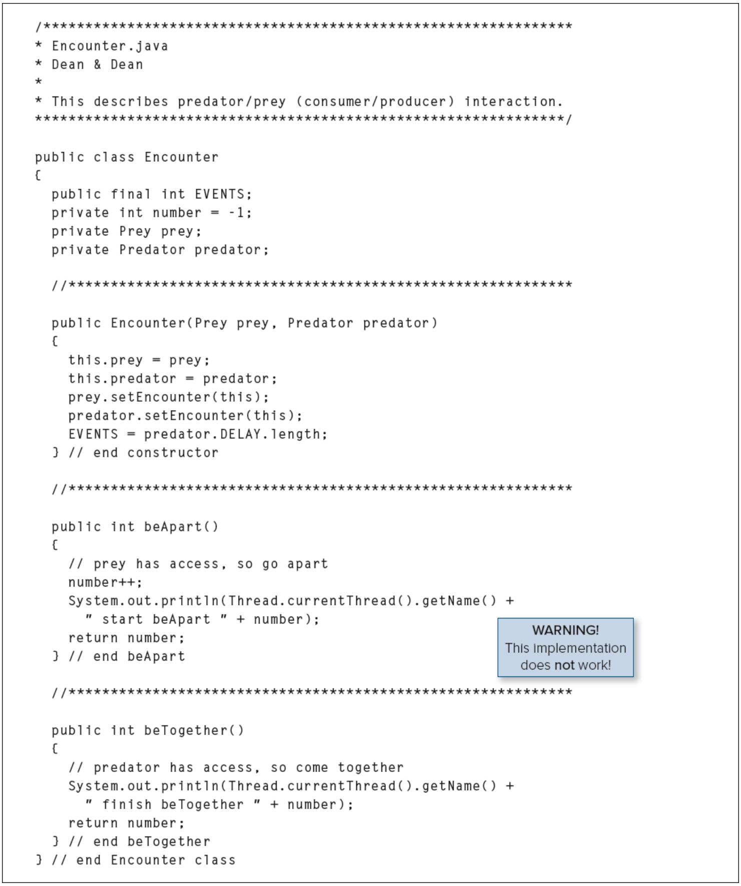
Now look at the beApart and beTogether methods. These represent the two phases of the ongoing encounter relationship. The beApart method describes a long quiescent period in which the predator rests and hunts. It's called by the Prey class in Figure A9.2. The beTogether method describes a short violent period in which the predator finds prey, attacks, and eats part of the prey. It's called by the Predator class in Figure A9.3.

As they appear in Figure A9.4, these two methods don't do very much. The beApart method updates the cycle number. Then it prints the name of the thread that called it and the cycle number’s current value. The beTogether method prints the name of the thread that called it and the cycle number’s current value. In a more complete model, the beTogether method would also call Predator and Prey methods to change the masses of these objects for the time they were apart. Then it would calculate the change in weights in the violent together period.



```
/*************************************************************
* Predator.java
* Dean & Dean
*
* This models predators (consumers), who desire encounters.
*************************************************************/

public class Predator extends Thread
{
  // delay times in milliseconds
  public final long[] DELAY = {2347, 1325, 1266, 3534};
  private Encounter encounter;

  //**********************************************************

  public Predator ()
  {
    super ("predator");
  } // end constructor

  //**********************************************************

  public void setEncounter(Encounter encounter)
  {
    this.encounter = encounter;
  } // end setEncounter

  //**********************************************************

  public void run()
  {
    int i;

    for (i=0; i<DELAY.length; i++)
    {
      try
      {
        Thread.sleep(DELAY[i]);          // rest & hunt
      }
      catch (Exception e) { }
      encounter.beTogether();            // eat prey
    }
    System.out.println(getName() + " run finished.");
  } // end run
} // end Predator class
```

Figure A9.3 Class describing a predator (consumer) which seeks prey This is driven by the class in Figure A9.1.



```
/***************************************************************
* Encounter.java
* Dean & Dean
*
* This describes predator/prey (consumer/producer) interaction.
***************************************************************/

public class Encounter
{
  public final int EVENTS;
  private int number = -1;
  private Prey prey;
  private Predator predator;

  //************************************************************

  public Encounter(Prey prey, Predator predator)
  {
    this.prey = prey;
    this.predator = predator;
    prey.setEncounter(this);
    predator.setEncounter(this);
    EVENTS = predator.DELAY.length;
  } // end constructor

  //************************************************************

  public int beApart()
  {
    // prey has access, so go apart
    number++;
    System.out.println(Thread.currentThread().getName() +
      " start beApart " + number);
    return number;
  } // end beApart

  //************************************************************

  public int beTogether()
  {
    // predator has access, so come together
    System.out.println(Thread.currentThread().getName() +
      " finish beTogether " + number);
    return number;
  } // end beTogether
} // end Encounter class
```

WARNING!
This implementation does not work!

Figure A9.4 Inadequate implementation of Encounter class Because the prey and predator threads run in parallel but only the predator thread contains delays, they do not interleave properly. The prey thread finishes quickly, whereas even the first output of the predator method is delayed until much later.

If you run the program in Figures A9.1, A9.2, A9.3, and A9.4, this is what you'll get:

Output:

```
prey start beApart 0
prey start beApart 1
prey start beApart 2
prey start beApart 3
prey run finished.
predator finish beTogether 3
predator finish beTogether 3
predator finish beTogether 3
predator finish beTogether 3
predator run finished.
```

Is this what you want? No! Because the prey and predator threads run in parallel but only the predator thread contains delays, they do not interleave properly. The prey thread finishes quickly, whereas even the first output of the predator thread does not occur until much later.

Synchronization

When different threads access a common object, they should be *synchronized*. We synchronize them relative to the common object by including the synchronized modifier in the heading of any common-object method that might be called by a thread that must be synchronized. In addition, we use a *semaphore* to give access to only one thread at a time—and *block* (temporarily stop) all other threads.

Figure A9.5a contains a corrected version of the first part of the Encounter class of Figure A9.4. In this part of the corrected class definition, the only thing different is the addition of another instance variable, a boolean semaphore that indicates which thread currently has access to the common object. This additional declaration appears in bold-face type.

Figure A9.5b contains a corrected

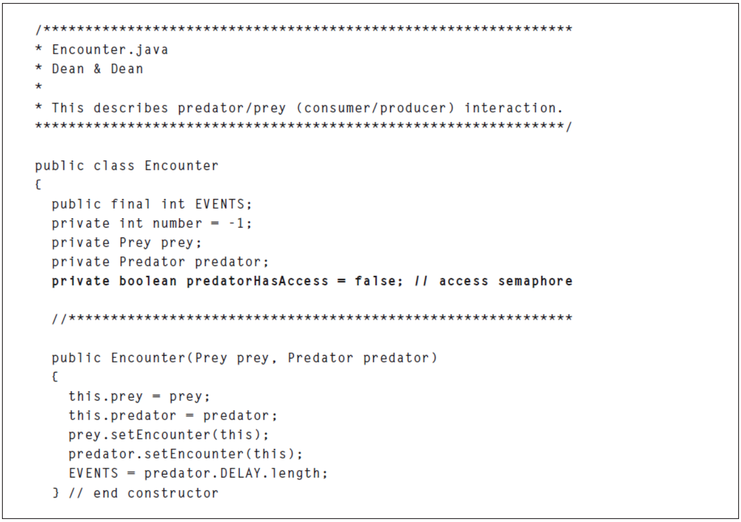
version of the second part of the Encounter class of Figure A9.4. All the new code appears in bold-face type. This shows what you do to synchronize multiple threads. First, include the synchronized modifier in the heading of those methods you want to synchronize. Then, at the start of each of those methods, put a while loop with a try block that contains the simple statement:

```
wait();
```

This statement blocks access to the rest of that method, so you should make the while condition true when you want to block access. Finally, insert a pair of special statements right before the return. The first of these special statements should set the phase of the semaphore to make the preceding while condition true. The second of these statements should be:

```
notifyAll();
```

In both cases the while loop’s condition is the phase of the semaphore that blocks the calling thread. If this condition is true when an external thread calls the method, flow goes immediately to the wait - statement4, and this blocks the execution of the calling thread at that point in its execution. That thread stays in this blocked state until it receives a “wake-up” system call initiated by another thread’s execution of the notifyAll method, at which time it starts running again—from the place where it was blocked. If the program is written correctly, the condition in the while loop of exactly one of the synchronized methods is false. When this particular method is called, flow jumps over the while loop to the subsequent executable code.



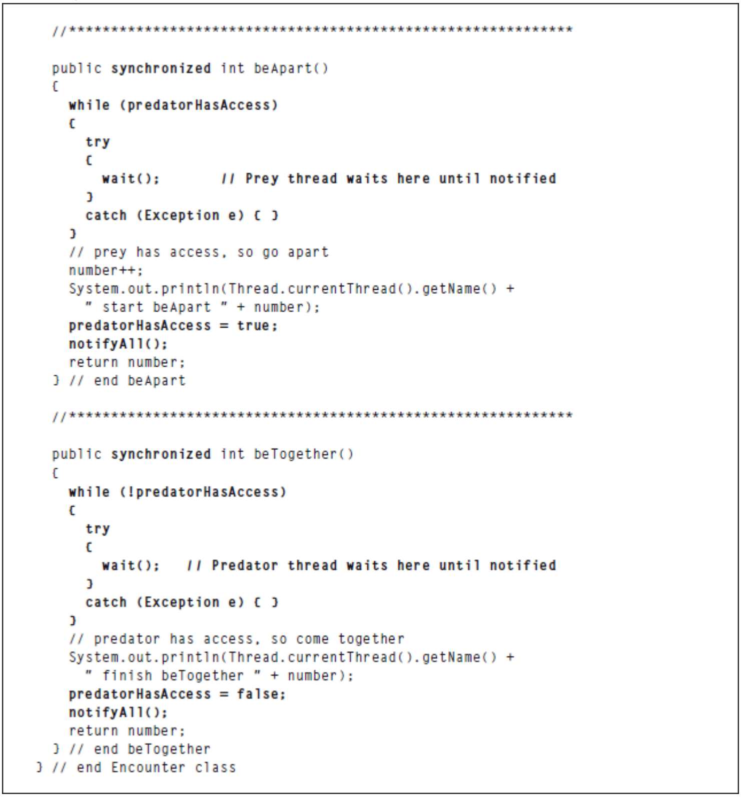
```
/*******************************************************************
* Encounter.java
* Dean & Dean
*
* This describes predator/prey (consumer/producer) interaction.
*******************************************************************/

public class Encounter
{
  public final int EVENTS;
  private int number = -1;
  private Prey prey;
  private Predator predator;
  private boolean predatorHasAccess = false; // access semaphore

  //*****************************************************************

  public Encounter(Prey prey, Predator predator)
  {
    this.prey = prey;
    this.predator = predator;
    prey.setEncounter(this);
    predator.setEncounter(this);
    EVENTS = predator.DELAY.length;
  } // end constructor
```

Figure A9.5a Corrected version of the Encounter class—Part A

For example, when in Figure A9.2 the prey thread first calls encounter.beApart, Figure A9.5a's predatorHasAccess semaphore is false. So in the beApart method in Figure A9.5b, prey thread execution jumps over the while loop and prints the output:

prey start beApart 0

Then the execution changes the predatorHasAccess semaphore to true, calls notifyAll, and returns. The next time the prey thread calls encounter.beApart, the true value of the while condition causes execution of the wait statement, and this blocks the prey thread at that point.



```
  //************************************************************

  public synchronized int beApart()
  {
    while (predatorHasAccess)
    {
      try
      {
        wait();       // Prey thread waits here until notified
      }
      catch (Exception e) { }
    }
    // prey has access, so go apart
    number++;
    System.out.println(Thread.currentThread().getName() +
      " start beApart " + number);
    predatorHasAccess = true;
    notifyAll();
    return number;
  } // end beApart

  //************************************************************

  public synchronized int beTogether()
  {
    while (!predatorHasAccess)
    {
      try
      {
        wait();   // Predator thread waits here until notified
      }
      catch (Exception e) { }
    }
    // predator has access, so come together
    System.out.println(Thread.currentThread().getName() +
      " finish beTogether " + number);
    predatorHasAccess = false;
    notifyAll();
    return number;
  } // end beTogether
} // end Encounter class
```

Figure A9.5b Corrected version of the Encounter class—Part B

Meanwhile (as soon as it is started), the predator thread begins running in parallel with the prey thread. When in

Figure A9.3 the predator thread first calls its sleep method, it stops execution for 2.347 seconds. During most of this time delay, both threads are blocked, and neither is executing. When the 2.347-second time delay expires, the predator thread automatically wakes up, jumps over the catch, and calls encounter.beTogether. Long before this time, the prey thread changed the -predatorHasAccess semaphore in Figure A9.5a to true, so in Figure A9.5b the predator thread execution jumps over beTogether's while loop and prints the output:

```
predator finish beTogether 0
```

Then the execution changes the predatorHasAccess semaphore to false, calls notifyAll, and returns. Back in the run method of Figure A9.3, the predator thread enters the second iteration of the for loop and goes to sleep again in its second time delay.

Meanwhile, the previous predator thread's notifyAll call re-activates the waiting prey thread and allows it to continue with the while loop in the beApart method in Figure A9.5b. This time, when execution returns to the while condition, it finds that the predatorHasAccess semaphore value is false. This allows it to escape from the while loop and print the output:

```
prey start beApart 1
```

Then, as before, it changes the predatorHasAccess semaphore to true, calls notifyAll, and returns.

This alternation between being apart and being together continues until (in the run method of Figure A9.2) number == encounter.EVENTS - 1, which terminates the prey thread. A little later (in the run method of Figure A9.3) i == DELAY.length, which terminates the predator thread. Using the corrected version of the Encounter class that appears in Figures A9.5a and A9.5b, the program output looks like this:

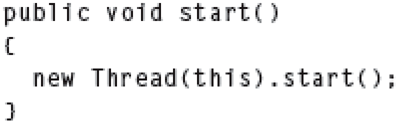
```
Output:
prey start beApart 0
predator finish beTogether 0
prey start beApart 1
predator finish beTogether 1
prey start beApart 2
predator finish beTogether 2
prey start beApart 3
prey run finished.
predator finish beTogether 3
predator run finished.
```

We encourage you to run this program yourself to get a physical sense of the interaction between the time-delay and wait operations.

---

[1] A “lightweight process” has its own “program counter” and its own “stack,” but otherwise it has normal access to

the rest of the program in which it exists. A thread’s program counter keeps track of where the execution is, and the stack remembers how to return from function calls. Whenever a thread temporarily stops, the computer takes a snapshot of that thread’s program counter and stack, and this enables the execution to re-start exactly where it left off when the thread starts running again.



```
public void start()
{
  new Thread(this).start();
}
```

[2] If your class already implements some class other than Thread, you can make it work like a thread by also implementing the Runnable interface. To start the run method of a class that implements Runnable but does not extend Thread, your class should also include a start method that does this:

[3] An InterruptedException is thrown when the current thread is sleeping

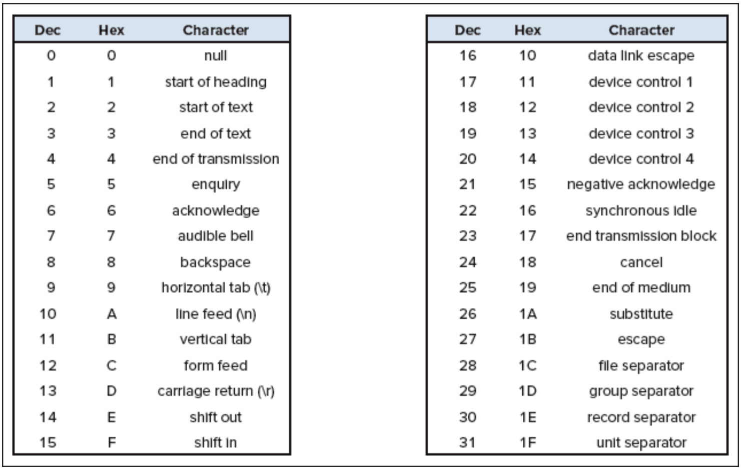
and another thread prematurely wakes it by calling its interrupt method, but our program never uses the interrupt method.

[4] The wait method is inherited by all objects from the Object class, and it throws an InterruptedException (just like the sleep method) if the waiting thread is interrupted while it's waiting. We must put the wait call in a try block because (as indicated previously) the exception it might throw is a checked exception, even though we never create the condition that throws that exception.

ASCII Character Set

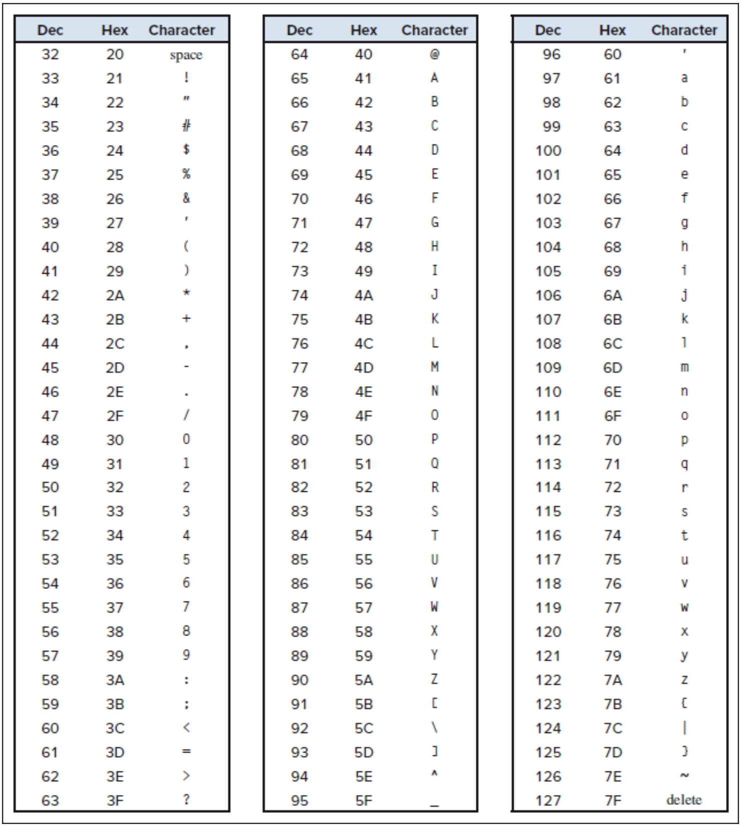
Java uses the Unicode standard to define all the characters that can be processed by a Java program. The Unicode standard is huge—it allows up to 1,112,064 characters. For an overview of Unicode, see Section 12.16 in Chapter 12. The first 128 characters of the Unicode character set are particularly important because they include the characters in the Latin alphabet—*A* through *Z*—plus other common characters, such as digits and punctuation marks. Those 128 characters used to be the only characters available to most programming languages, and they formed the American Standard Code for Information Interchange (ASCII) character set. See Figures A1.1a and A1.1b, which show the ASCII characters and their associated decimal and hexadecimal values.

In the 1960s, when the ASCII character set was born, the characters' positions were chosen carefully in order to satisfy the most people and build support for a common standard. For example, the *#, $,* and *%* characters were chosen to be next to each other because they appear next to each other on most typewriters. In Figure A1.1b, note that *#, $,* and *%* are at positions 35, 36, and 37, respectively. Also, note in Figure A1.1b that the first uppercase letter, *A*, is at hexadecimal position 41, and the first lowercase letter, *a*, is at hexadecimal position 61. To change case, you can simply add or subtract hexadecimal 20. The people who created the ASCII character set were thinking in hex!



Dec	Hex	Character	Dec	Hex	Character
0	0	null	16	10	data link escape
1	1	start of heading	17	11	device control 1
2	2	start of text	18	12	device control 2
3	3	end of text	19	13	device control 3
4	4	end of transmission	20	14	device control 4
5	5	enquiry	21	15	negative acknowledge
6	6	acknowledge	22	16	synchronous idle
7	7	audible bell	23	17	end transmission block
8	8	backspace	24	18	cancel
9	9	horizontal tab (\t)	25	19	end of medium
10	A	line feed (\n)	26	1A	substitute
11	B	vertical tab	27	1B	escape
12	C	form feed	28	1C	file separator
13	D	carriage return (\r)	29	1D	group separator
14	E	shift out	30	1E	record separator
15	F	shift in	31	1F	unit separator

Figure A1.1a First 32 characters in the ASCII character set

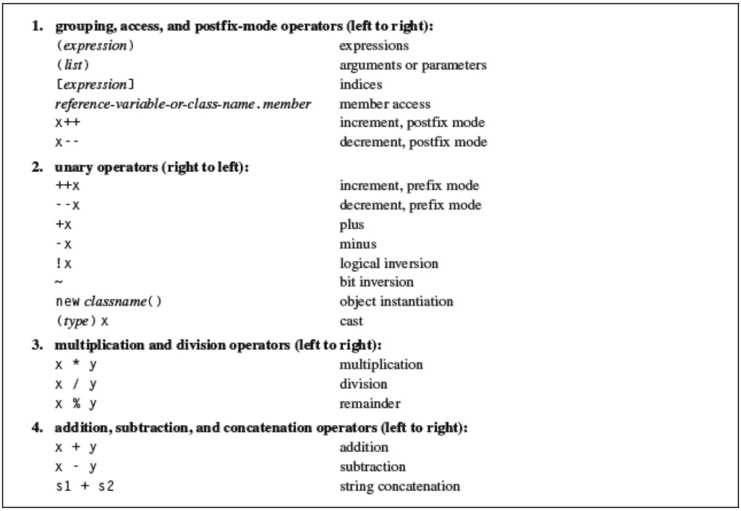


Dec	Hex	Character	Dec	Hex	Character	Dec	Hex	Character
32	20	space	64	40	@	96	60	'
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	delete

Figure A1.1b Remaining characters in the ASCII character set

Operator Precedence

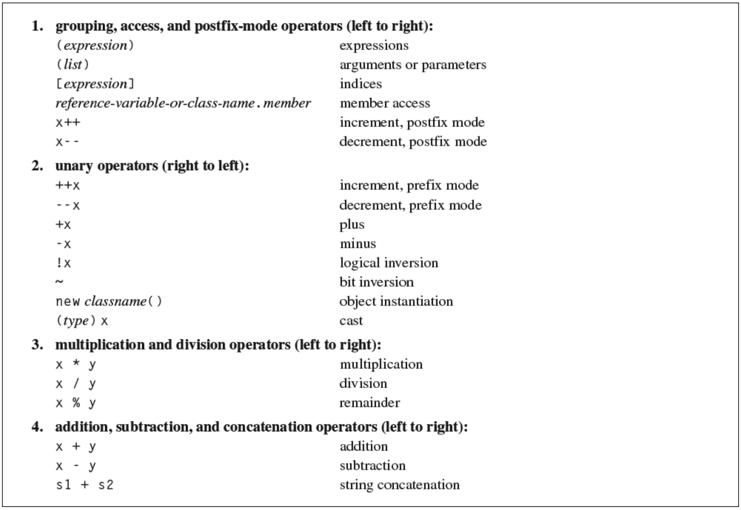
The table in Figures A2.1a and A2.1b shows operator precedence. The operator groups at the top of the table have higher precedence than the operator groups at the bottom of the table. All operators within a particular precedence group have equal precedence. If an expression has two or more same-precedence operators, then within that expression, those operators execute from left to right or right to left, as indicated in the group heading.



1. grouping, access, and postfix-mode operators (left to right):	
(*expression*)	expressions
(*list*)	arguments or parameters
[*expression*]	indices
reference-variable-or-class-name.*member*	member access
x++	increment, postfix mode
x--	decrement, postfix mode
2. unary operators (right to left):	
++x	increment, prefix mode
--x	decrement, prefix mode
+x	plus
-x	minus
!x	logical inversion
~	bit inversion
new *classname*()	object instantiation
(*type*) x	cast
3. multiplication and division operators (left to right):	
x * y	multiplication
x / y	division
x % y	remainder
4. addition, subtraction, and concatenation operators (left to right):	
x + y	addition
x - y	subtraction
s1 + s2	string concatenation

Figure A2.1a Operator precedence—part A

The unconditional operators in groups 8, 9, and 10 evaluate all operands, even though the final condition may be determined before all operands have been evaluated. The conditional operators in groups 11, 12, and 13 stop the evaluation process as soon as the final condition is determined. Groups 8, 9, 10, 11, and 12 describe expressions with two different types of operations, depending on the types of operators. If the operators are boolean, the operations are logical, with a true or false result for each case. If the operators are integers or characters, each bit evaluates separately.

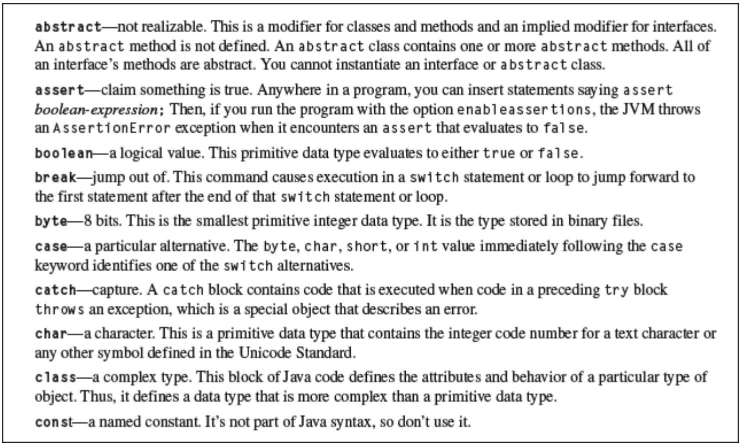


1. grouping, access, and postfix-mode operators (left to right):	
`(`*expression*`)`	expressions
`(`*list*`)`	arguments or parameters
`[`*expression*`]`	indices
reference-variable-or-class-name`.`*member*	member access
`x++`	increment, postfix mode
`x--`	decrement, postfix mode
2. unary operators (right to left):	
`++x`	increment, prefix mode
`--x`	decrement, prefix mode
`+x`	plus
`-x`	minus
`!x`	logical inversion
`~`	bit inversion
`new` *classname*`()`	object instantiation
`(`*type*`) x`	cast
3. multiplication and division operators (left to right):	
`x * y`	multiplication
`x / y`	division
`x % y`	remainder
4. addition, subtraction, and concatenation operators (left to right):	
`x + y`	addition
`x - y`	subtraction
`s1 + s2`	string concatenation

Figure A2.1b Operator precedence—part B

Java Keywords and Other Reserved Words

The table in Figures A3.1a, A3.1b, A3.1c, and A3.1d presents Java *keywords*. In general, Java keywords are words that you cannot use as identifiers in your programs because they have special meanings in the Java language. For example, if, while, and public are standard Java keywords. Two of Java's keywords, const and goto, are special in that they are not allowed to be used anywhere in a Java program—not as Java syntax and not as identifiers. In talking about words where you're constrained in how you can use them, we say those words are "reserved." In addition to Java's keywords, there are a few other words that are reserved. true, false, and null are reserved (you can't use them as identifiers), but they're classified as *literals*, not keywords. var is reserved (you can't use it as an enumerated type), but it's classified as a *reserved type name*, not a keyword.



`abstract`—not realizable. This is a modifier for classes and methods and an implied modifier for interfaces. An `abstract` method is not defined. An `abstract` class contains one or more `abstract` methods. All of an interface's methods are abstract. You cannot instantiate an interface or `abstract` class.

`assert`—claim something is true. Anywhere in a program, you can insert statements saying `assert` *boolean-expression*`;` Then, if you run the program with the option `enableassertions`, the JVM throws an `AssertionError` exception when it encounters an `assert` that evaluates to `false`.

`boolean`—a logical value. This primitive data type evaluates to either `true` or `false`.

`break`—jump out of. This command causes execution in a `switch` statement or loop to jump forward to the first statement after the end of that `switch` statement or loop.

`byte`—8 bits. This is the smallest primitive integer data type. It is the type stored in binary files.

`case`—a particular alternative. The `byte`, `char`, `short`, or `int` value immediately following the `case` keyword identifies one of the `switch` alternatives.

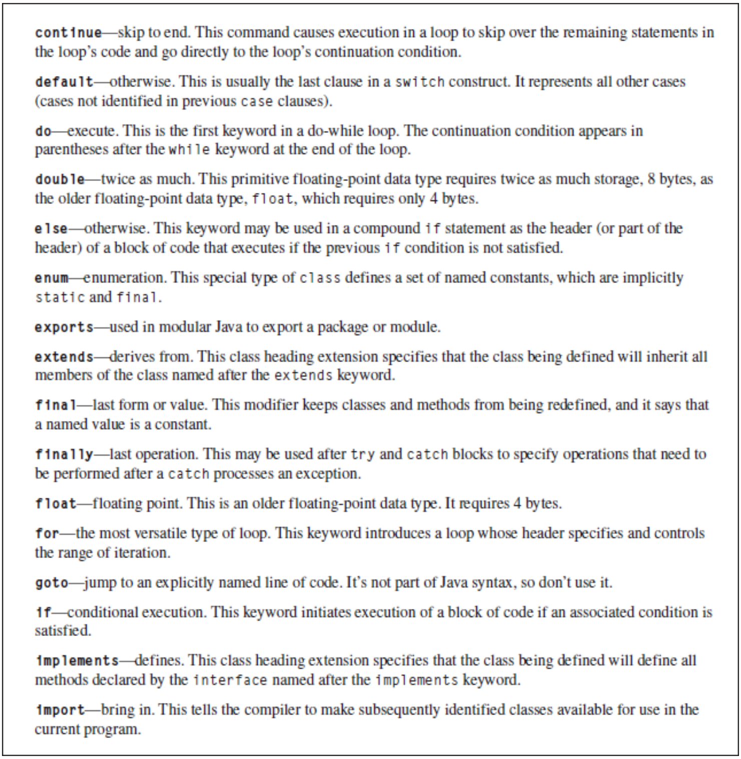
`catch`—capture. A `catch` block contains code that is executed when code in a preceding `try` block `throws` an exception, which is a special object that describes an error.

`char`—a character. This is a primitive data type that contains the integer code number for a text character or any other symbol defined in the Unicode Standard.

`class`—a complex type. This block of Java code defines the attributes and behavior of a particular type of object. Thus, it defines a data type that is more complex than a primitive data type.

`const`—a named constant. It's not part of Java syntax, so don't use it.

Figure A3.1a Keywords—part A



`continue`—skip to end. This command causes execution in a loop to skip over the remaining statements in the loop's code and go directly to the loop's continuation condition.

`default`—otherwise. This is usually the last clause in a `switch` construct. It represents all other cases (cases not identified in previous `case` clauses).

`do`—execute. This is the first keyword in a do-while loop. The continuation condition appears in parentheses after the `while` keyword at the end of the loop.

`double`—twice as much. This primitive floating-point data type requires twice as much storage, 8 bytes, as the older floating-point data type, `float`, which requires only 4 bytes.

`else`—otherwise. This keyword may be used in a compound `if` statement as the header (or part of the header) of a block of code that executes if the previous `if` condition is not satisfied.

`enum`—enumeration. This special type of `class` defines a set of named constants, which are implicitly `static` and `final`.

`exports`—used in modular Java to export a package or module.

`extends`—derives from. This class heading extension specifies that the class being defined will inherit all members of the class named after the `extends` keyword.

`final`—last form or value. This modifier keeps classes and methods from being redefined, and it says that a named value is a constant.

`finally`—last operation. This may be used after `try` and `catch` blocks to specify operations that need to be performed after a `catch` processes an exception.

`float`—floating point. This is an older floating-point data type. It requires 4 bytes.

`for`—the most versatile type of loop. This keyword introduces a loop whose header specifies and controls the range of iteration.

`goto`—jump to an explicitly named line of code. It's not part of Java syntax, so don't use it.

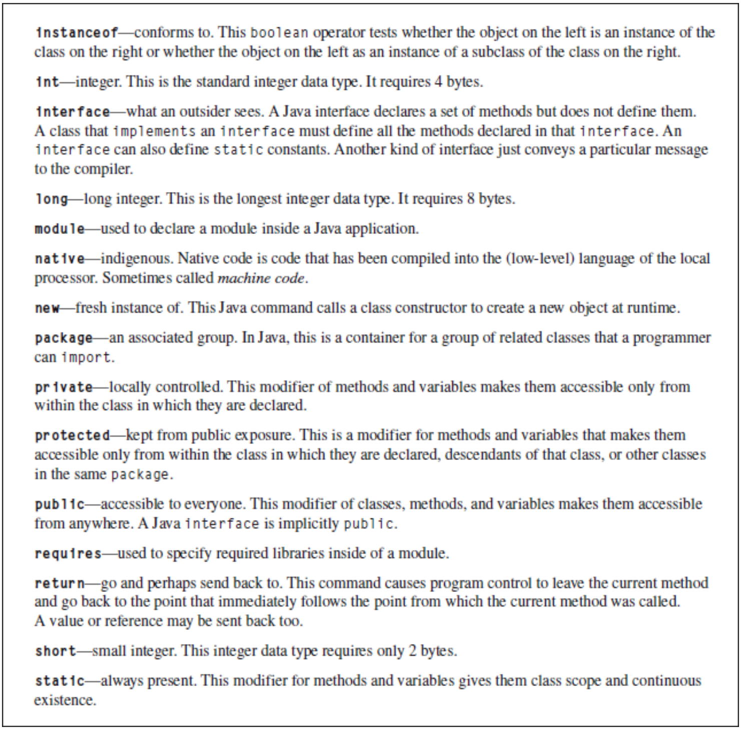
`if`—conditional execution. This keyword initiates execution of a block of code if an associated condition is satisfied.

`implements`—defines. This class heading extension specifies that the class being defined will define all methods declared by the `interface` named after the `implements` keyword.

`import`—bring in. This tells the compiler to make subsequently identified classes available for use in the current program.

Figure A3.1b Keywords—part B

Page 949



`instanceof`—conforms to. This `boolean` operator tests whether the object on the left is an instance of the class on the right or whether the object on the left as an instance of a subclass of the class on the right.

`int`—integer. This is the standard integer data type. It requires 4 bytes.

`interface`—what an outsider sees. A Java interface declares a set of methods but does not define them. A class that `implements` an `interface` must define all the methods declared in that `interface`. An `interface` can also define `static` constants. Another kind of interface just conveys a particular message to the compiler.

`long`—long integer. This is the longest integer data type. It requires 8 bytes.

`module`—used to declare a module inside a Java application.

`native`—indigenous. Native code is code that has been compiled into the (low-level) language of the local processor. Sometimes called *machine code*.

`new`—fresh instance of. This Java command calls a class constructor to create a new object at runtime.

`package`—an associated group. In Java, this is a container for a group of related classes that a programmer can `import`.

`private`—locally controlled. This modifier of methods and variables makes them accessible only from within the class in which they are declared.

`protected`—kept from public exposure. This is a modifier for methods and variables that makes them accessible only from within the class in which they are declared, descendants of that class, or other classes in the same `package`.

`public`—accessible to everyone. This modifier of classes, methods, and variables makes them accessible from anywhere. A Java `interface` is implicitly `public`.

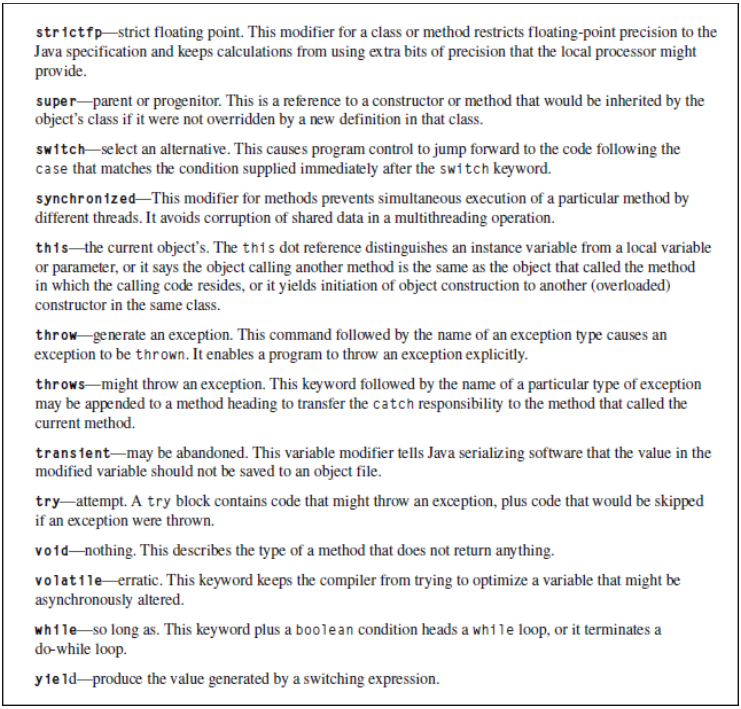
`requires`—used to specify required libraries inside of a module.

`return`—go and perhaps send back to. This command causes program control to leave the current method and go back to the point that immediately follows the point from which the current method was called. A value or reference may be sent back too.

`short`—small integer. This integer data type requires only 2 bytes.

`static`—always present. This modifier for methods and variables gives them class scope and continuous existence.

Figure A3.1c Keywords—part C



`strictfp`—strict floating point. This modifier for a class or method restricts floating-point precision to the Java specification and keeps calculations from using extra bits of precision that the local processor might provide.

`super`—parent or progenitor. This is a reference to a constructor or method that would be inherited by the object's class if it were not overridden by a new definition in that class.

`switch`—select an alternative. This causes program control to jump forward to the code following the `case` that matches the condition supplied immediately after the `switch` keyword.

`synchronized`—This modifier for methods prevents simultaneous execution of a particular method by different threads. It avoids corruption of shared data in a multithreading operation.

`this`—the current object's. The `this` dot reference distinguishes an instance variable from a local variable or parameter, or it says the object calling another method is the same as the object that called the method in which the calling code resides, or it yields initiation of object construction to another (overloaded) constructor in the same class.

`throw`—generate an exception. This command followed by the name of an exception type causes an exception to be `thrown`. It enables a program to throw an exception explicitly.

`throws`—might throw an exception. This keyword followed by the name of a particular type of exception may be appended to a method heading to transfer the `catch` responsibility to the method that called the current method.

`transient`—may be abandoned. This variable modifier tells Java serializing software that the value in the modified variable should not be saved to an object file.

`try`—attempt. A `try` block contains code that might throw an exception, plus code that would be skipped if an exception were thrown.

`void`—nothing. This describes the type of a method that does not return anything.

`volatile`—erratic. This keyword keeps the compiler from trying to optimize a variable that might be asynchronously altered.

`while`—so long as. This keyword plus a `boolean` condition heads a `while` loop, or it terminates a do-while loop.

`yield`—produce the value generated by a switching expression.

Figure A3.1d Keywords—part D

Packages and Modules

As you know, a *package* is a group of related classes. A *module* provides additional grouping capabilities—it groups related packages. In implementing a module, you specify dependencies. Specifically, you specify other modules that the new module depends on and the packages it makes available to other modules.

In this appendix, we start with a more in-depth look at packages, and then we describe how modules work. We then use packages and modules in the context of Java's new GUI platform, JavaFX. JavaFX is not part of the standard Java installation package. To write programs with JavaFX, you need to first download it from OpenJFX, at https://openjfx.io. The modular separation complicates compiling and execution somewhat, and this appendix explains how to perform these operations in the new environment. Then, it explains how to package and modularize code you create for use by others. Modules make Java more versatile and improve security, but they add programming complexity.

Java API Packages

Before Java 11, whenever you downloaded a version of Java from a supplier like Oracle, you got approximately 300 different packages as part of the Java Development Kit (JDK). The installation automatically made those packages part of your Java environment. Each of those packages contained a collection of Java API interfaces and classes.

Figure A4.1 shows just a few of the hundreds of Java API packages contained in seven file directories under the root names: com, java, javafx, javax, jdk, netscape, and org. These root directories (in dashed boxes) do not implement packages by themselves because they do not contain any Java interfaces or classes. They just contain subordinate directories that implement packages. The package name in each subordinate box is like the full path to the subdirectory containing that package's contents, except the package name uses periods instead of path slashes.

The packages identified in Figure A4.1 are some of the packages discussed in this book and its website. Hierarchical packaging helps people find particular classes they need to use. It's OK for several different classes to have the same class name if they are in different packages. Consequently, encapsulating small groups of classes into individual packages allows us to reuse a given class name in different contexts. Also, each package prevents outside access to all protected members of its classes.

To make the classes in a particular

package available to a program you're working on, you must import that package, as in these statements, which import the java.util and javafx.scene packages:

```
import java.util.*;
import javafx.scene.*;
```

Figure A4.1 shows several packages three levels down from the top. For example, in the javafx tree, under javafx.scene consider the javafx.scene.layout package. Importing the javafx.scene package in the statement above provides access to all classes in the javafx.scene package itself, but it does not also provide access to packages under it. In other words, it does not also import the javafx.scene.layout package. If you also need access to classes in the javafx.scene.layout package, you also must import that package explicitly by adding this third import statement:

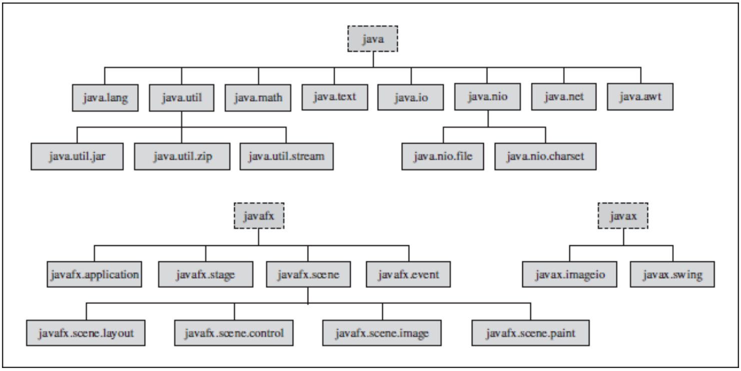
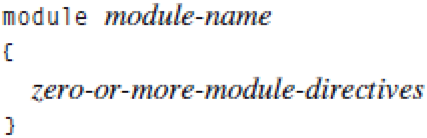

java
java.lang
java.util
java.math
java.text
java.io
java.nio
java.net
java.awt
java.util.jar
java.util.zip
java.util.stream
java.nio.file
java.nio.charset
javafx
javax
javafx.application
javafx.stage
javafx.scene
javafx.event
javax.imageio
javax.swing
javafx.scene.layout
javafx.scene.control
javafx.scene.image
javafx.scene.paint

Figure A4.1 Abbreviated examples of Java API package hierarchies

```
import javafx.scene.layout.*;
```

Java API Modules

In the culmination of Project Jigsaw, Java SE 9 and Java JDK 9 introduced another layer of encapsulation called *modules*. Each Java module may contain any number of Java packages. Modules improve security by providing strong encapsulation. To this end, each module contains a special *module descriptor*, which is the compiled version of a little program called module-info.java that looks like this:



```
module module-name
{
  zero-or-more-module-directives
}
```

The header's *module-name* establishes the module's name, using the same convention as packages use (periods instead of slashes) to describe where the module is in a Java-module directory structure.

Optional module directives include:

- exports *package-name*;
- makes a particular contained package accessible to all other modules
- exports *package-name* to *comma-separated-list-of-module-names*;
- makes a particular contained package accessible to specified other modules
- requires *module-name*;

identifies another module that contains accessible packages this module needs to use

Modular organization makes it feasible to bundle application software with different combinations of standard and custom code. This enhances Java’s versatility. It enables diverse operating systems to download zipped versions of appropriate application code onto diverse hardware—ranging from large supercomputers to small mobile devices and programmable controllers.

Go to Oracle’s Java API website. In the header click OVERVIEW. You should see a list of about 22 modules in the JAVA SE (java.) category and about 37 modules in the JDK (jdk.) category. Most of the classes discussed in this book and its website (other than JavaFX GUI classes) are contained in just one of these modules, java.base. Clicking on the java.base module link takes you to a web page that states that java.base holds the foundational

APIs of the Java SE Platform. The java.base module exports about 52 packages, including java.io, java.lang, java.math, and java.util. Access to this module is automatic.

For JavaFX APIs, go to https://openjfx.io, and under JavaDoc click the Visit button. This should take you to a web page that lists seven modules, including javafx.base, javafx.graphics, and javafx.controls.

The javafx.base module “Defines the base APIs for the JavaFX UI toolkit, including APIs for bindings, properties, collections, and events.” It contains the javafx.event package, which contains the ActionEvent class.

The javafx.graphics module “Defines the core scenegraph APIs for the JavaFX UI toolkit (such as layout containers, application lifecycle, shapes, transformations, canvas, input, painting, image handling, and effects), as well as APIs for animation, CSS, concurrency, geometry, printing, and windowing.” It contains packages like javafx.application, javafx.stage, and javafx.scene, as well as javafx.geometry, javafx.scene.layout, javafx.scene.paint, javafx.scene.image, and javafx.scene.shape. The javafx.graphics module requires javafx.base and also specifically exports all of javafx.base’s packages in what is called an *indirect export*. So if another module requires javafx.graphics, then it indirectly gains accesses to all packages in javafx.base.

The javafx.controls module “Defines the UI controls, charts, and skins that are available for the JavaFX UI toolkit.” It defines just four packages, but one of these is the important javafx.scene.control package, which itself contains more than one hundred classes, including Alert, Button, CheckBox, ChoiceDialog, ComboBox, Label, RadioButton, TextField, and

ToggleGroup. This module requires javafx.base and javafx.graphics and provides indirect exports of all of their packages. So if another module requires javafx.controls, then it indirectly gains access to all packages in javafx.base and javafx.graphics.

Installation and Use of JavaFX

Before you install JavaFX, you'll need to have installed the core Java JDK, which you can get from https://www.oracle.com/technetwork/java/javase/downloads/. If you're using Windows 10, click the Start button and type "advanced system settings." Click Environment Variables, add or edit a JAVA_HOME variable to make it equal the directory containing your Java JDK software. Also edit your computer's Path variable so that it includes the path to the bin subdirectory of the new JAVA_HOME directory.1

Also download the JavaFX JDK from

a site like https://openjfx.io. You may add this JavaFX JDK to the directory that contains the Java JDK or you may put it anywhere else. In either case you must remember the path to the JavaFX SDK, because you must explicitly supply the path to its lib subdirectory every time you compile or execute any program that uses Java 11 (or later) and one or more JavaFX modules.

One way to do this job is to create an environmental variable like PATH_TO_FX and set it equal to the path to the lib subdirectory of the directory containing the JavaFX SDK. Then, if you want to compile a program that needs the JavaFX graphics module, go to the directory that contains your source code, and instead of entering something like javac HelloFX.java, enter:

```
>javac -p %PATH_TO_FX% --add-modules javafx.graphics HelloFX.java
```

where -p is an abbreviation for –-module-path, and PATH_TO_FX is an environment variable that holds the location of your JavaFX files. You could use this same environmental variable again to execute the compiled program like this:

```
>java -p %PATH_TO_FX% --add-modules javafx.graphics HelloFX
```

The -p %PATH_TO_FX% is a particular kind of *command option*[2] called a *path directive*. This directive tells the compiler where to find the modules the compiler needs to do its job. If you need multiple modules on different paths, precede each subsequent path specification with a semicolon and no space after that semicolon, and precede each subsequent module specification with a comma and no space after that comma. The --add-modules javafx.graphics is an *add-modules directive*. It identifies particular modules the compiler needs in addition to those that come with core Java software.

If your program also needs to use the javafx.controls module, instead of using:

```
--add-modules javafx.graphics
```

you could use:

```
--add-modules javafx.graphics,javafx.controls
```

But because the javafx.controls module automatically provides everything that's in the javafx.graphics module, it's easier to specify the javafx.controls module by itself.

Another way to do the job is to put all of the -p and --add-modules information in a separate text file, perhaps called fx.txt, and save this text file in a convenient place. If that place is the directory that contains the source code, you could compile from that directory with:

```
>javac @fx.txt HelloFX.java
```

By default, compiled code goes into the current directory, so you could execute from that same directory with:

```
>java @fx.txt HelloFX
```

The fx.txt file's @ prefix tells javac and javafx to use the text file's contents as directives.

When you use a file like fx.txt to make one or more additional modules available, you don't need a PATH_TO_FX system variable, because you can save the path it specifies in that fx.txt file. If the path to JavaFX software in your computer is C:\Java11\javafx-sdk-11.0.1\lib and your program needs to access both javafx.graphics and javafx.controls, the contents of your fx.txt file could be like this:

```
-p C:\Java11\javafx-sdk-11.0.1\lib --
add-modules javafx.controls
```

This file's contents must be all on one line, and there must be NO carriage returns!

Instead of providing a separate copy of fx.txt in each local directory, you can put just one copy of it at a central location, like the root of a drive that contains and will contain all of your source and object code. Then you can compile and execute from any subordinate directory with commands like these:

```
>javac @\fx.txt HelloFX.java
>java @\fx.txt HelloFX
```

Package Specification and Custom Module Creation and Use

Java lets you specify packaging for classes you create. We'll illustrate this with examples that begin a process of organizing this book's Java code. All of our packages will have a common root directory called ipwj, which represents this book’s title, *Introduction to Programming with Java.* Under this common root will be subdirectories for each of the book’s 19 chapters—ch01, ch02, … ch19. Code for each program that is not an optional end-of-chapter GUI program will go directly into its corresponding chapter directory. When a particular chapter contains one or more optional GUI sections, code in those sections will go into a subdirectory called gui.

An ipwj structure containing all of our source code will be a child of a parent directory called src, which also might contain other source code structures for other books. In parallel with the src directory will be another directory called mods. This mods directory will contain java modules with package subdirectories that contain compiled versions of the programs in the src directory tree.

Into the source code for every class, we will insert—as the very first statement—a specification of the package in which that class belongs. Then, we'll compile the source code in a way that automatically puts the compiled code into its specified package and then puts that package into a module in the mods directory.

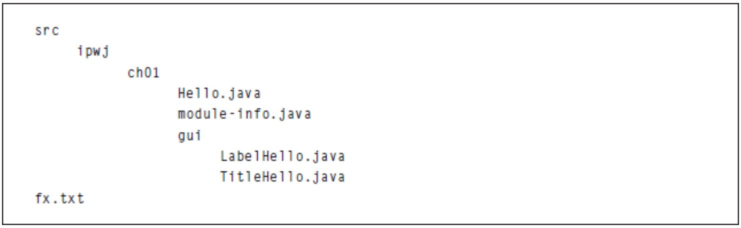
Our book's Chapter 1 contains three programs, Hello.java in Figure 1.6, TitleHello.java in Figure 1.12, and LabelHello.java in Figure 1.13. The Hello.java program is a non-GUI program. To specify a package for it, insert the following statement at the very beginning of its source code:

```
package ipwj.ch01;
```

The second and third of Chapter 1's three programs are GUI programs in Section 1.10 GUI Track: Hello World (Optional). Since these two programs each require JavaFX, we specify another package for each of them by inserting the following statement at the very beginning of each of their source codes:

```
package ipwj.ch01.gui;
```

Figure A4.2 shows a directory tree that contains Chapter 1’s source code and reflects the packaging specified by the initial package statements we will have inserted into each of this chapter’s programs. The ch01 subdirectory contains the only Chapter 1 non-GUI program, Hello.java. The subordinate gui subdirectory contains the two Chapter 1 GUI programs, TitleHello.java and LabelHello.java.

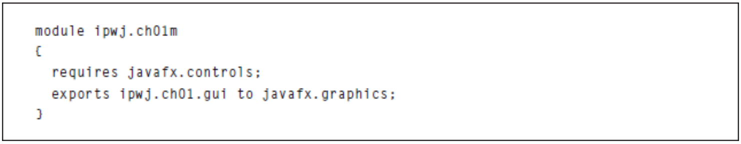


```
src
    ipwj
         ch01
              Hello.java
              module-info.java
              gui
                   LabelHello.java
                   TitleHello.java
fx.txt
```

Figure A4.2 Directory structure for source code of all Chapter 1 programs

In addition to Chapter 1’s only non-

GUI program, Hello.java, our ch01 subdirectory also contains a special program called module-info.java, shown in Figure A4.3. The ipwj.ch01m that appears after the initial term, module, is the desired module name. Like a package name, this module name uses periods to suggest a descending hierarchical structure.[3] For our module name we append “m” to the ch01 directory name because a module name should not end in a digit and because this appendage helps us distinguish our module names from our package names.



```
module ipwj.ch01m
{
  requires javafx.controls;
  exports ipwj.ch01.gui to javafx.graphics;
}
```

Figure A4.3 module-info.java program for modules that include JavaFX GUI programs

In Figure A4.3’s module-info program, ipwj.ch01.gui identifies a (local) package, and javafx.controls and javafx.graphics identify (remote) modules. If none of a module's programs employ JavaFX GUI—as is the case for ipwj.ch04m—or if a module is to be used only with a Java release before release 11, that module's module-info.java program wouldn't need the indicated requires and exports statements. It could simply proclaim the module name, like this:

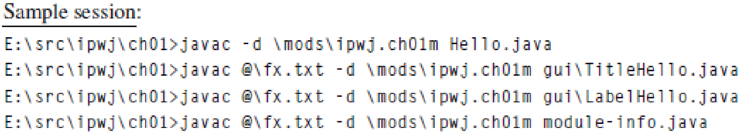
```
module ipwj.ch04m { }
```

Figure A4.2 shows an fx.txt file in parallel with a src directory at the root of the hierarchy. Since almost all of the book's programs that need an fx.txt file can use exactly the same one, it's most convenient to put just one copy at a single easily accessible place—the root of the drive containing the book's source code. Then, assuming we were currently in a subdirectory like src\ipwj\ch01\gui, we could compile the TitleHello.java program with a command that uses a single slash to specify the drive-root location of the fx.txt file like this:

>javac @\fx.txt -d \mods \ipwj.ch01m TitleHello.java

In this command, the -d \mods \ipwj.ch01m is a *destination directive*. It tells the computer where to put the compiled file.4

Now navigate to the ch01 directory, which contains fx.txt, module-info.java, and all of Chapter 1's non-GUI source code. Then, compile all of the java programs in this directory tree, with commands that use either forward or reverse slashes except for period(s) in desired module names and source code file names, like this:



Sample session:

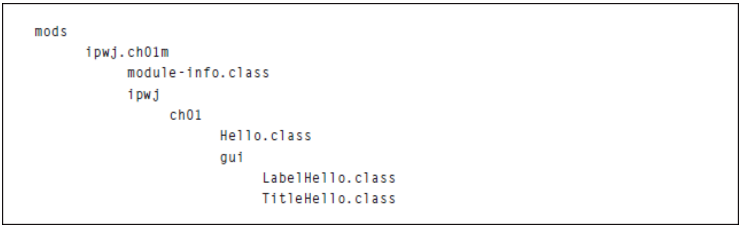
```
E:\src\ipwj\ch01>javac -d \mods\ipwj.ch01m Hello.java
E:\src\ipwj\ch01>javac @\fx.txt -d \mods\ipwj.ch01m gui\TitleHello.java
E:\src\ipwj\ch01>javac @\fx.txt -d \mods\ipwj.ch01m gui\LabelHello.java
E:\src\ipwj\ch01>javac @\fx.txt -d \mods\ipwj.ch01m module-info.java
```

If a particular command's specified destination structure is not already present, its -d destination directive automatically creates that structure and then puts the compiled form of the command's source code into it. Subsequent commands add the compiled form of their source code into matching existing structures.

The compiling operations automatically create a subdirectory under mods whose name is the same as the module name, ipwj.ch01m. Whereas our source code directory structure had a separate ch01 subdirectory nested under its ipwj directory, the corresponding module structure combines these in just one directory called ipwj.ch01m. That's because the compile command's destination directive used a period instead of a slash in its ipwj.ch01m. The preceding sample session's final javac command puts the compiled code for the module-info program into this new ipwj.ch01m directory, as shown in

Figure A4.4.

Page 957



```
mods
        ipwj.ch01m
              module-info.class
              ipwj
                    ch01
                           Hello.class
                           gui
                                 LabelHello.class
                                 TitleHello.class
```

Figure A4.4 Directory structure for modularized version of compiled Hello programs

The other javac commands in the preceding sample session put the compiled code for all of the other programs at places in a subordinate directory structure that conform to those programs' package statements—package ipwj.ch01 for the Hello program and package ipwj.ch01.gui for the TitleHello and LabelHello programs. Notice how package-statement periods in source code correspond to subdirectories in compiled-code.

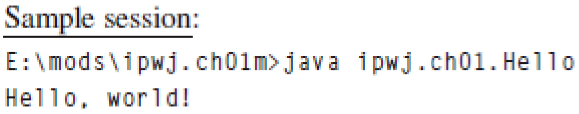
The compiling sequence is sensitive to dependencies. If you try to compile module-info.java before compilation of one of the two GUI programs has created the full module structure, the compiler complains like this:

```
error: package is empty or does not exist: ipwj.ch01.gui
```

Once a previously compiled module-info has established a requires condition, if you try to recompile non-GUI code like Hello.java without @\fx.txt (as in the preceding sample session), the compiler complains like this:

```
error: module not found: javafx.controls
```

Now, any user who has a copy of the compiled structure in Figure A4.4 can navigate to the ipwj.ch01m module directory[5] and run the Hello program like this:

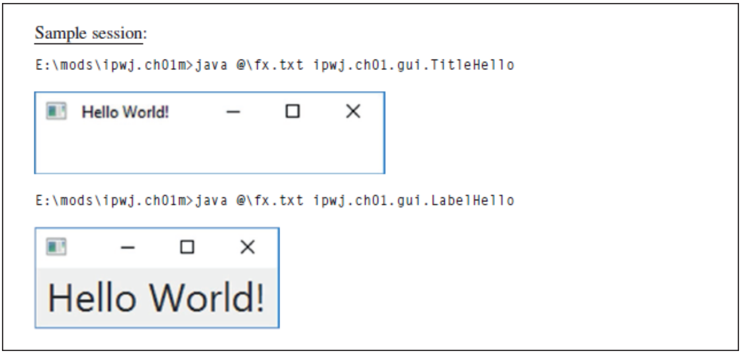


Sample session:

```
E:\mods\ipwj.ch01m>java ipwj.ch01.Hello
Hello, world!
```

Notice how the execution command specifies class location within the module's package hierarchy with periods instead of slashes.6

Suppose that same user creates an fx.txt file like that in Figure A4.2, but with but with its -p directive specifying the path to the JavaFX module on that user's computer. If that fx.txt file is at the root of the drive containing the directory structure in Figure A4.4, the user can run the module's two GUI programs as Figure A4.5.



Sample session:

```
E:\mods\ipwj.ch01m>java @\fx.txt ipwj.ch01.gui.TitleHello
```

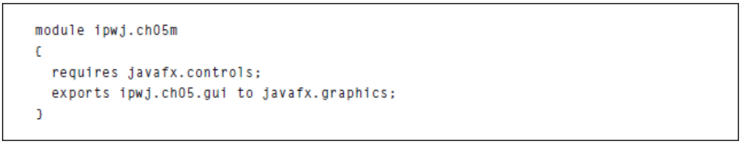
```
E:\mods\ipwj.ch01m>java @\fx.txt ipwj.ch01.gui.LabelHello
```

Figure A4.5 Execution of HelloTitle and HelloLabel programs

Again, notice how the execution commands specify class location within the module's package hierarchy with periods instead of slashes.

Accessing an External Resource

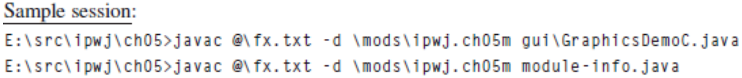
Next, suppose we have a program that needs to access an external resource, like the GraphicsDemoC program in Section 5.9's Figure 5.14 accesses dolphinsC.jpg. For this program to go in a module, it must be in a package, so we specify that package by inserting this additional statement at the very beginning of its source code: package ipwj.ch05.gui; Then we create source code for a module-info program like that in Figure A4.6. This is the same as that in Figure A4.3 except for two substitutions of ch05 for ch01.



```
module ipwj.ch05m
{
  requires javafx.controls;
  exports ipwj.ch05.gui to javafx.graphics;
}
```

Figure A4.6 module-info.java program for modules that include JavaFX GUI programs

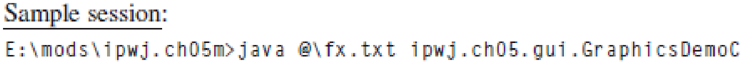
Again, with an appropriate fx.txt file at the root of the drive containing the source code, we navigate to the directory containing the source code and compile like this:



Sample session:

```
E:\src\ipwj\ch05>javac @\fx.txt -d \mods\ipwj.ch05m gui\GraphicsDemoC.java
E:\src\ipwj\ch05>javac @\fx.txt -d \mods\ipwj.ch05m module-info.java
```

Now, alongside the module-info.class in the module directory, ipwj.ch05m, we paste a copy of the dolphinsC.jpg resource file. Then from that same directory we execute[7] the compiled code like this:



Sample session:

```
E:\mods\ipwj.ch05m>java @\fx.txt ipwj.ch05.gui.GraphicsDemoC
```

Putting Regular Driver and Driven Classes in the Same Package

Next, let’s see how to put regular (not JavaFX) driver and driven classes together in the same package. We’ll start with an example from Chapter 6—the Mouse class in Section 6.3’s

Figure 6.4 and the MouseDriver class in Section 6.4’s Figure 6.5. At the very top of the source code for both of these programs, insert the following additional statement: package ipwj.ch06; In the source code hierarchy in Figure A4.2, under ipwj, add the subdirectory, ch06, and insert the revised Mouse.java and MouseDriver.java programs in this new subdirectory.

From within this E:\src\ipwj\ch06 subdirectory, compile the driven class, Mouse.java, with this command:

```
>javac -d \mods\ipwj.ch06m Mouse.java
```

This command enhances the \mods directory structure in Figure A4.4 to include the new path \mods\ipwj.ch06m\ipwj\ch06. The command explicitly creates the ipwj.ch06m directory, and Mouse’s package ipwj.ch06; statement implicitly creates the two directories below it. Then the command compiles Mouse.class and inserts it into the lowest ch06 subdirectory.

Then, again from within the E:\src\ipwj\ch06 subdirectory, compile the driver class, MouseDriver.java, with this command:

```
>javac -cp \mods\ipwj.ch06m -d \mods\ipwj.ch06m MouseDriver.java
```

The -cp is an abbreviation for -classpath or --class-path. It tells the compiler where to find another class that the current class needs to access. This operation adds MouseDriver.class alongside Mouse.class in the previously created \mods\ipwj.ch06m\ipwj\ch06 directory.

Now navigate to the \mods\ipwj.ch06m directory, and execute the driver with this command:

```
>java ipwj.ch06.MouseDriver
```

The executing MouseDriver calls the

previously compiled Mouse in the same package. Notice that the execution command uses the package specification's periods instead of slashes to call the driver class from the ipwj.ch06m directory, which is outside the part of the directory tree that corresponds to the program's package specification. Trying to execute from inside the package subtree does NOT work:

Sample session:

```
E:\mods\ipwj.ch06m\ipwj\ch06>java MouseDriver
Error: Could not find or load main class MouseDriver
```

Putting JavaFX Driver and Driven Classes in the Same Package

Page 960

Next, let's see how to put JavaFX driver and driven classes together in the same package. Chapter 8's GUI track (Problem Solving with CRC Cards) presents java code for two classes, CRC_Card.java and CRCDriver.java. At the very top of the source code for each of these programs, insert the following additional statement: package ipwj.ch08.gui; Then, in the source code hierarchy in Figure A4.2, under ipwj, add the subtree, ch08\gui, and insert the revised CRC_Card.java and CRCDriver.java programs in the new gui subdirectory.

From within this new E:\src\ipwj\ch08\gui subdirectory, compile the driven class, CRC_Card.java, with this command:

```
>javac @\fx.txt -d \mods\ipwj.ch08m CRC_Card.java
```

In the tree shown in Figure A4.4, under mods, this will create a new subtree, ipwj.ch08m\ipwj\ch08\gui, and into the new gui subdirectory it will insert CRC_Card.class. Then, from within this E:\src\ipwj\ch08\gui subdirectory, compile the driver class, CRCDriver.java, with this command:

```
>javac @\fx.txt -cp \mods\ipwj.ch08m -d \mods\ipwj.ch08m CRCDriver.java
```

This operation will add CRCDriver.class alongside CRC_Card.class in the new gui subdirectory.

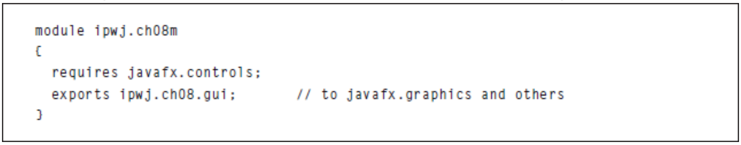
Now navigate to the \mods\ipwj.ch08m directory, and execute the driver with this command:

```
>java @\fx.txt ipwj.ch08.gui.CRCDriver
```

The executing CRCDriver calls the previously compiled CRC_Card in the same package. This displays a blank version of the CRC card in Figure 8.18, and the user can fill it in as shown in that figure or in any other way.

The above operations create the same modular directory structure as the structures created previously for our Chapter 1 and Chapter 5 GUI programs. But what's created is not really a module yet because it doesn't have its own module-info program. To complete the modularization, into the \src\ipwj\ch08 directory, insert the source code in Figure A4.7. Then, from this same directory, compile the new module-info source code with the following command:

```
>javac @\fx.txt -d \mods \ipwj.ch08m module-info.java
```



```
module ipwj.ch08m
{
  requires javafx.controls;
  exports ipwj.ch08.gui;       // to javafx.graphics and others
}
```

Figure A4.7 module-info.java program for modules that include JavaFX GUI programs

This module-info program differs from that in Figure A4.6 in two ways. First, the module name has changed from ipwj.ch05m to ipwj.ch08m. Second, the exports directive is more general. Whereas Chapter 5's module exported its ipwj.ch05.gui package to just the javafx.graphics module, Chapter 8's module will export its ipwj.ch08.gui package to any other module that wants it. This more general exportation from the ipwj.ch08m is not needed for anything we have done up to this point, but it is needed for what we'll do next.

Using a Driver That's in a Different Module

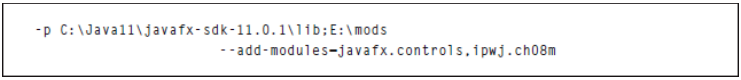
Finally, let's see how to implement and use an external driver in a different module. At the end of Chapter 10 in Section 10.12 GUI Track: Second Iteration of Problem Solving with CRC Cards (Optional), there is a second CRC_Card driver, CRCDriver2. This second CRC_Card driver also needs to access the CRC_Card class in the ipwj.ch08m module. But CRCDriver2 will be in a different module. That’s why we generalized the exports directive in Figure A4.7.

At the very top of the source code for Chapter 10’s CRCDriver2 program, insert the following two additional statements:

```
package ipwj.ch10.gui;
import ipwj.ch08.gui.CRC_Card;
```

Then, in the source code hierarchy shown Figure A4.2, under ipwj, add the subtree, ch10\gui, and insert the revised CRCDriver2.java program in the new gui subdirectory.

Because the driven class is now in a different module, we need additional command options beyond those in our generic fx.txt file. Figure A4.8 shows the contents of the required new command-option file, fx10.txt. This specifies an additional module path by appending ;E:\mods to the previous path, and it specifies an additional module by appending ,ipwj.ch08 to the previous module. There must be no space after the semicolon and comma that precede these additional specifications, and although this file's contents appear on two lines in Figure A4.8, there must be no carriage returns.



```
-p C:\Java11\javafx-sdk-11.0.1\lib;E:\mods
                                --add-modules=javafx.controls,ipwj.ch08m
```

Figure A4.8 Contents of our fx10.txt file

Although shown as split onto two lines at an internal space, that split must be just a space and not a carriage return. That is, this file's contents must be all on one line, with no carriage returns.

To avoid cluttering the drive's root directory with specialized command-option files, put this specialized fx10.txt file into the E:\src\ipwj\ch10\gui directory. Then, from within this directory, compile the revised second driver with this command:

```
>javac @fx10.txt -cp \mods\ipwj.ch08m -d \mods\ipwj.ch10m CRCDriver2.java
```

In the tree shown in Figure A4.4, under the mods, this operation creates another new subtree, ipwj.ch10m\ipwj\ch10\gui, and inserts CRCDriver2.class into its final gui directory.

Next, paste a copy of the fx10.txt file into the newly created \mods\ipwj.ch10m directory. While in this directory, execute the new driver with the following command:

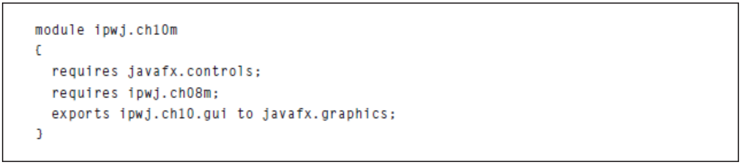
```
>java @fx10.txt ipwj.ch10.gui.CRCDriver2 Class1 Class2
```

This should display blank cards for Class1 and Class2 in the upper left corner and a dialog window in the center of the screen.

Although the name of the current directory, ipwj.ch10m, looks a module name, it’s not yet a module because it does not have a compiled module-info file. To finish modularization, go back to the \src\ipwj\ch10 directory, create the module-info file shown in Figure A4.9, and while still in that directory issue the following command:

```
>javac @gui\fx10.txt -d \mods\ipwj.ch10m module-info.java
```

This puts the new module-info.class file into the \mods\ipwj.ch10m directory alongside the previously copied fx10.txt file. And the preceding execution of CRCDriver2 still works.



```
module ipwj.ch10m
{
  requires javafx.controls;
  requires ipwj.ch08m;
  exports ipwj.ch10.gui to javafx.graphics;
}
```

Figure A4.9 module-info.java program for Chapter 10 module

Java Coding-Style Conventions

This appendix describes Java coding-style conventions. Most of these guidelines are widely accepted. -However, alternative guidelines do exist in certain areas. The coding conventions presented in this document are for the most part a simplified subset of the archived coding conventions presented on the Java Code Conventions web page:

http://www.oracle.com/technetwork/java/codeconventions-150003.pdf

If you have a style question that is not addressed in this document, refer to Oracle's coding conventions document or go to Google's Java Style Guide at https://google.github.io/styleguide/javaguide.html.

While reading the following sections, refer to the example program in the last section. You can mimic the style in that example.

Prologue

1. Put this prologue section at the top of the file:

```
/
***********************************
* <filename>
* <programmer's name>
*
* <file description>
***********************************
```

2. Include a blank line below the prologue section.

Section Delimiting

1. After state variable definitions and between constructor and method definitions, enter a line of stars, like this:

```
//
***********************************
```

Leave a blank line above and below this line of stars.

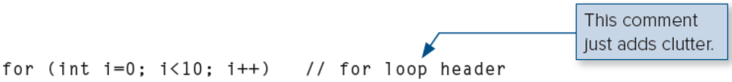
2. Within a large constructor or method, insert blank lines between logical sections of code. For example, unless the loops are small and intimately related, enter a blank line between the end of one loop and the beginning of another loop.

Embedded Comments

1. Provide comments for code that might be confusing to someone reading your program for the first time. Assume the reader understands Java syntax.

Page 964

2. Do not comment code that is obvious. For example, this comment is unnecessary and therefore exhibits poor style:



```
for (int i=0; i<10; i++)    // for loop header
```

This comment just adds clutter.

3. Write your programs with clear, self-documenting code in order to reduce the need for comments. For example, use mnemonic (descriptive) identifier names.

4. Always include a single space between the // and the comment text.

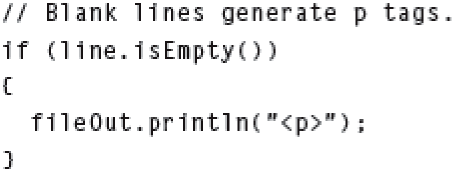
5. The comment's length determines its format.

- If the comment will occupy more than one line, use complete lines, like this:

  // This is a block comment. Use it for comments that

// occupy more than one line.
Note the alignment for /'s
// and words.

- If a comment is to reside on a line by itself, position it above the line of code it describes. Indent the // the same as the described line of code. Include a blank line above the comment line. Here's an example:



```
// Blank lines generate p tags.
if (line.isEmpty())
{
  fileOut.println("<p>");
}
```

- Many comments are small enough to fit in space to the right of the code they describe. Whenever possible, all such comments should start in the same column, as far to the right as - possible. The following example demonstrates proper positioning for short comments:

```
double testScores = new double[80]; // test scores for one class
int student;

...
while (testScores[student] >= 0)    // negative score quits
{
  testScores[student] = score;
  ...
```

6. Provide an end comment for each closing brace that is a significant number of lines (five or more?) down from its matching opening brace. For example, note the // end for row and // end getSum comments in the following method definition:

```
public double getSum(double table[][], int rows, int cols)
{
  double sum = 0.0;

  for (int row=0; row<rows; row++)
  {
    for (int col=0; col<cols; col++)
    {
      sum += table[row][col];
    } // end for col
  } // end for row

  return sum;
} // end getSum
```

Variable Declarations

1.Normally, you should declare only one variable per line. For example:

```
double avgScore;      // average score on the test
int numOfStudents;    // number of students in the class
```

Exception:

If several variables are intimately related, it is acceptable to declare them together on one line. For example:

Page 965

```
int x, y, z;          // coordinates for a point
```

2. Normally, you should include a comment for each variable declaration line.

Exception:

Don’t include a comment for names that are obvious (i.e., studentId) or standard (i.e., i for a for loop index variable, ch for a character variable).

Braces That Surround One Statement

1. For if, else, for, while, and do constructs that execute only one statement, it’s good practice to treat that one statement as though it were a compound statement and enclose it in braces, like this:

```
for (int i=0; i<scores.length; i++)
{
  sumOfSquares += scores[i] * scores[i];
}
```

2. Exception:

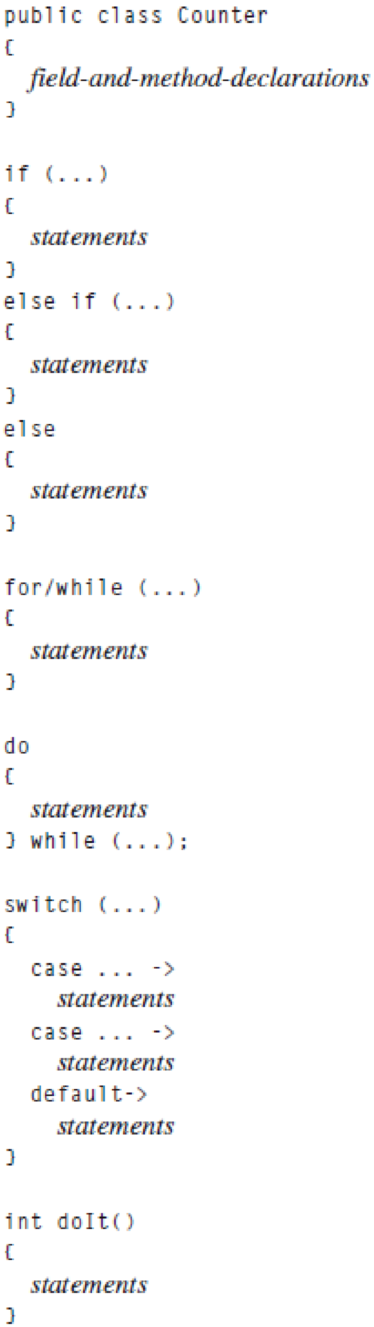
If it would be illogical to add another statement to the construct at a later time, you may omit the curly braces when the omission improves readability. For example, an experienced programmer might write something like this:

```
for (; num>=2; num--)
  factorial *= num;
```

Placement of Braces

1. Place opening and closing braces on lines by themselves, such that the braces are aligned with the line above the opening brace. For do loops, put the while condition on the same line as the closing brace.

2. Examples:

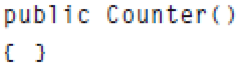


```
public class Counter
{
   field-and-method-declarations
}

if (...)
{
   statements
}
else if (...)
{
   statements
}
else
{
   statements
}

for/while (...)
{
   statements
}

do
{
   statements
} while (...);

switch (...)
{
   case ... ->
      statements
   case ... ->
      statements
   default->
      statements
}

int doIt()
{
   statements
}
```

3. Brace alignment is a

contentious issue. Oracle’s Java Code Conventions website recommends putting the opening curly brace at the end of the previous line. This is one place where our conventions diverge from Oracle’s conventions. We recommend that you put the opening curly brace on its own line because that helps make compound statements stand out.

4. For empty-bodied constructors, place the opening and closing braces on the same line and separate them with a space, like this:



```
public Counter()
{ }
```

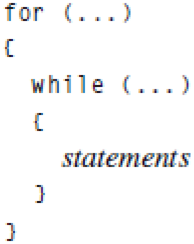
The else if Construct

1. If the body of an else is just another if, form an else if construct (put the else and the if on the same line). See the above brace placement section for an example of a proper else if construct.

Alignment and Indentation

1. Align all code that is logically at the same level. See the above brace placement section for examples of proper alignment.

2. Indent all code that is logically inside other code. That is, for nested logic, use nested indentation. For example:

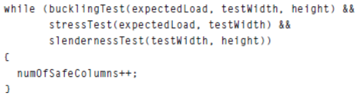


```
for (...)
{
  while (...)
  {
    statements
  }
}
```

3. You may use an indentation width of two to five spaces. Once you choose an indentation width, you should stick with it. Use the same indentation width throughout your program.

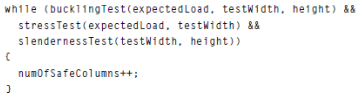
4. When a statement is too long to fit on one line, write it on multiple lines such that the continuation lines are indented appropriately. If the long statement is followed by a single statement that is logically inside of the long statement, use braces to enclose the single statement. Use either of the following techniques to indent continuation lines:

- Indent to a column position such that similar entities are aligned. In the example below, the entities that are aligned are the three method calls:



```
while (bucklingTest(expectedLoad, testWidth, height) &&
       stressTest(expectedLoad, testWidth) &&
       slendernessTest(testWidth, height))
{
  numOfSafeColumns++;
}
```

- Indent the same number of spaces as all other indents. For example:



```
while (bucklingTest(expectedLoad, testWidth, height) &&
  stressTest(expectedLoad, testWidth) &&
  slendernessTest(testWidth, height))
{
  numOfSafeColumns++;
}
```

Multiple Statements on One Line

Page 968

1. Normally, each statement should be put on a separate line.

Exception:

If statements are intimately related and very short, it is acceptable (but not required) to put them together on one line. For example:

a++; b++; c++;

2. For assignment statements that are intimately related and use the same assigned value, it is acceptable (but not required) to combine them into one assignment statement. For example:

x = y = z = 0;

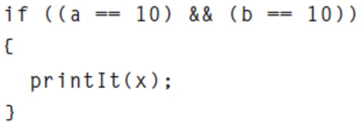
Spaces within a Line of Code

1. Never put a space at the left of a semicolon.

2. Parentheses:

- Never enter a space on the inside of enclosing parentheses.
- If the entity to the left of a left parenthesis is an operator or a construct keyword (if, switch, etc.), then precede the parenthesis with a space.
- If the entity to the left of a left parenthesis is a method name, then do not precede the parenthesis with a space.

For example:



```
if ((a == 10) && (b == 10))
{
  printIt(x);
}
```

3. Operators:

- Normally, an operator should be surrounded by spaces. For example:

```
if (x >= 3 && x <= 7)
{
  y = (a + b) / 2;
}
```

- Special cases:
    - Complex expressions:

      —Within an inner component of a complex expression, do not surround the inner component's operators with spaces.

      —Two common occurrences of complex expressions are conditional expressions and for loop headers. See the examples below.

    - Dot operator—no spaces at its left or right.
    - Unary operators—no

space between unary operator and its associated operand.

For example:

```
if (zeroMinimum)
{
   x = (x<0 ? 0 : x);
}

while (list1.row != list2.row)
{
   statements
}

for (int i=0,j=0; i<=bigI; i++,j++)
{
   statements
}
```

Shortcut Operators

1. Use increment and decrement operators instead of their equivalent longer forms. For example:

Do not use	Use this	
`x = x + 1`	`x++ or ++x`	(depending on the context)
`x = x - 1`	`x-- or --x`	(depending on the context)

2. Use compound assignments instead of their equivalent longer forms. For example:

Do not use	Use this
`x = x + 5`	`x += 5`
`x = x * (3 + y)`	`x *= 3 + y`

Naming Conventions

1. Use meaningful names for your identifiers.

2. For named constants, use all uppercase letters. If there are multiple words, use underscores to separate the words. For example:

```
public static final int SECONDS_IN_DAY = 86400;
private final int ARRAY_SIZE;
```

3. For class names (and their associated constructors), use uppercase for the first letter and lowercase for all other letters. If there are multiple words in the class name, use uppercase for the first letter of all words. For example:

```
public class InnerCircle
{
  public InnerCircle(radius)
  {
    statements
  }
```

4. For all identifiers other than constants and constructors, use all lowercase letters. If there are multiple words in the identifier, use uppercase for the first letter of all words that follow the first word. For example:

```
double avgScore;     // average score on the test
int numOfStudents;   // number of students in the class
```

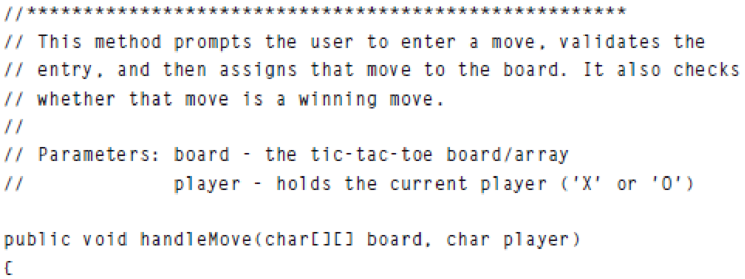
Method and Constructor Organization

1. Normally, each method definition should be preceded by these items:

- a blank line
- a line of *'s
- a blank line

- a description of the purpose of the method
- a blank line
- parameter descriptions (for non-obvious parameters)
- a blank line

Ideally, all method parameters should use descriptive enough names so that the purpose of each parameter is inherently obvious. However, if this is not the case, then include a list of parameters and their descriptions in a method prologue above the method heading. For example, in a tic-tac-toe program, a method that handles a player's move would be relatively complicated and would require a method prologue like this:

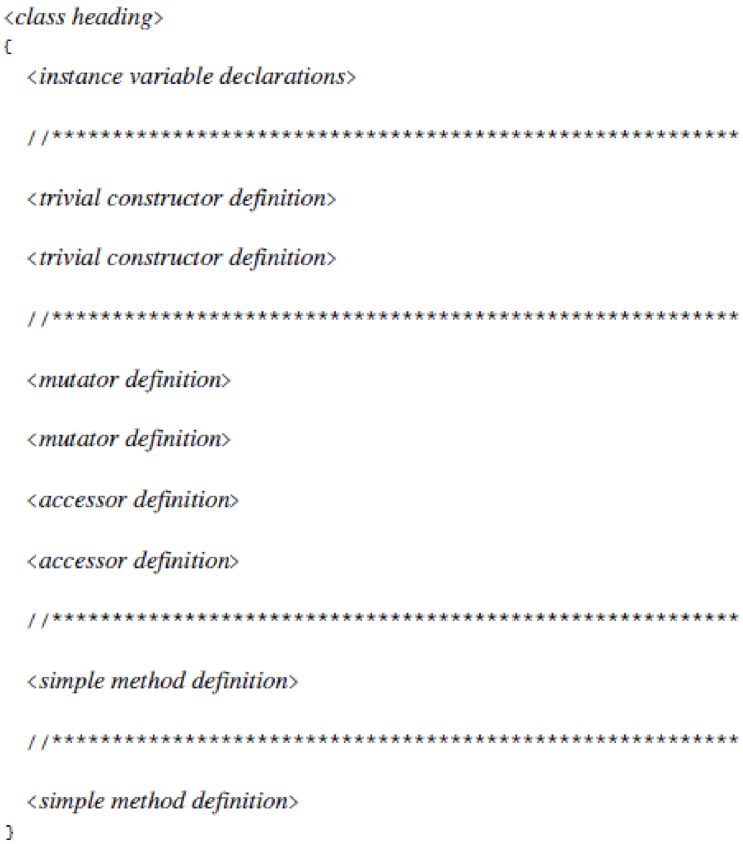


```
//*****************************************************
// This method prompts the user to enter a move, validates the
// entry, and then assigns that move to the board. It also checks
// whether that move is a winning move.
//
// Parameters: board - the tic-tac-toe board/array
//             player - holds the current player ('X' or 'O')

public void handleMove(char[][] board, char player)
{
```

Assuming you describe instance and static variables when you declare them, you should not provide prologues for “trivial” accessors, mutators, and constructors that just read or write instance and static variables. On the other hand, if a mutator performs validation on a parameter prior to assigning it to its associated instance variable, then it is not trivial, and you should include a prologue with it. The same reasoning applies to a constructor. A simple-assignment constructor should not have a prologue. A validation constructor should have a prologue.

2. In the interest of grouping similar things together, you should omit asterisk lines between trivial constructors, and you should omit asterisk lines between mutators and accessors.

Assume that a class contains two trivial constructors, several mutator and accessor methods, and two other simple methods. Here’s the framework for such a class:



```
<class heading>
{
   <instance variable declarations>

   //*********************************************************

   <trivial constructor definition>

   <trivial constructor definition>

   //*********************************************************

   <mutator definition>

   <mutator definition>

   <accessor definition>

   <accessor definition>

   //*********************************************************

   <simple method definition>

   //*********************************************************

   <simple method definition>
}
```

In the above framework, note that there are no descriptions for trivial constructors, accessors, or mutators, or for simple methods. Note also that there is a line of asterisks above the first mutator, but not above the subsequent mutator and accessors. Those omissions help to make a program more readable by grouping similar things together. Also, note that there are no comments above each of the two simple methods at the bottom of the class, but there are lines of asterisks.

3. Place local variable declarations immediately below the method heading. Do not place local variable declarations within the executable code.

Exception: Declare a for loop index variable within its for loop header.

Class Organization

1. Each of your classes may contain the following items (in the following order):

- **a)** class prologue section
- **b)** import statements
- **c)** static constants
- **d)** instance constants

e) static variables

f) instance variables

g) abstract methods

h) constructors

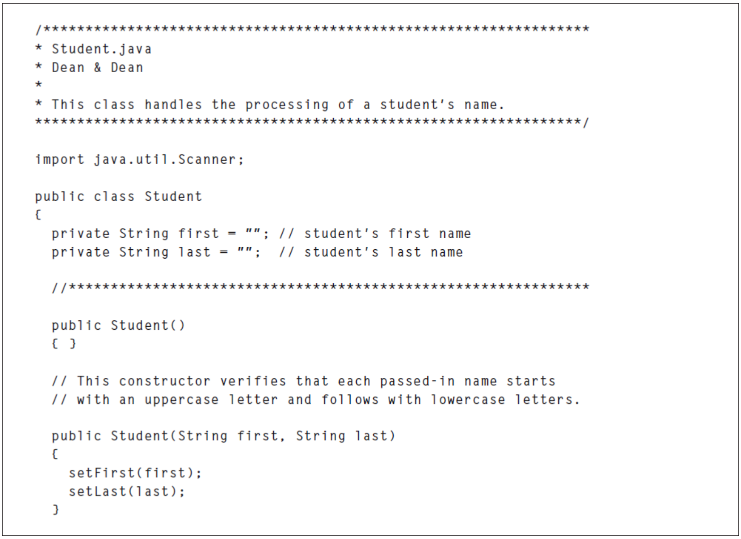
i) instance methods

j) static methods

2. Normally, you should place a main method and any of its helper methods in its own separate driver class. But it's sometimes appropriate to include a short main method within the class that it drives as an embedded testing tool. Put such a method at the end of the class definition.

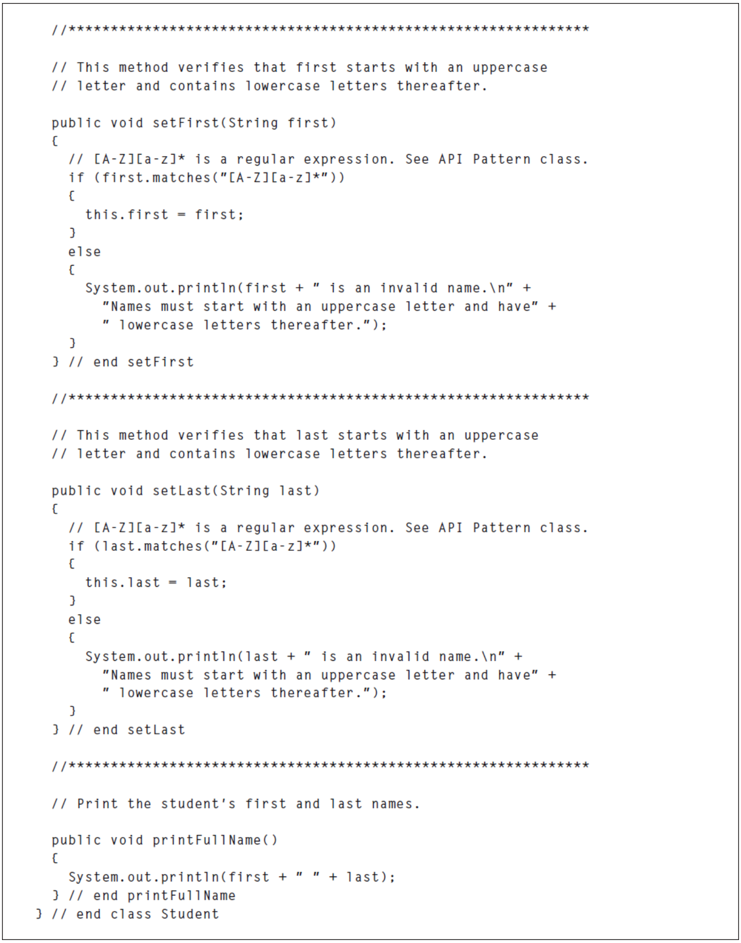
Sample Java Program

For example, look at the coding style in the Student program in Figures A5.1a, A5.1b, and A5.2.



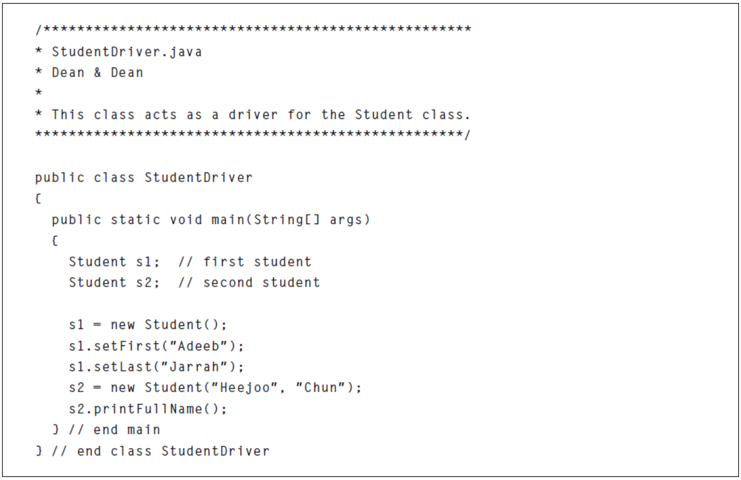
```
/*******************************************************************
* Student.java
* Dean & Dean
*
* This class handles the processing of a student's name.
*******************************************************************/

import java.util.Scanner;

public class Student
{
  private String first = ""; // student's first name
  private String last = "";  // student's last name

  //*****************************************************************

  public Student()
  { }

  // This constructor verifies that each passed-in name starts
  // with an uppercase letter and follows with lowercase letters.

  public Student(String first, String last)
  {
    setFirst(first);
    setLast(last);
  }
```

Figure A5.1a Student class, used to illustrate coding conventions—part A



```
  //**************************************************************

  // This method verifies that first starts with an uppercase
  // letter and contains lowercase letters thereafter.

  public void setFirst(String first)
  {
    // [A-Z][a-z]* is a regular expression. See API Pattern class.
    if (first.matches("[A-Z][a-z]*"))
    {
      this.first = first;
    }
    else
    {
      System.out.println(first + " is an invalid name.\n" +
        "Names must start with an uppercase letter and have" +
        " lowercase letters thereafter.");
    }
  } // end setFirst

  //**************************************************************

  // This method verifies that last starts with an uppercase
  // letter and contains lowercase letters thereafter.

  public void setLast(String last)
  {
    // [A-Z][a-z]* is a regular expression. See API Pattern class.
    if (last.matches("[A-Z][a-z]*"))
    {
      this.last = last;
    }
    else
    {
      System.out.println(last + " is an invalid name.\n" +
        "Names must start with an uppercase letter and have" +
        " lowercase letters thereafter.");
    }
  } // end setLast

  //**************************************************************

  // Print the student's first and last names.

  public void printFullName()
  {
    System.out.println(first + " " + last);
  } // end printFullName
} // end class Student
```

Figure A5.1b Student class, used to illustrate coding conventions—part B



```
/****************************************************
* StudentDriver.java
* Dean & Dean
*
* This class acts as a driver for the Student class.
****************************************************/

public class StudentDriver
{
  public static void main(String[] args)
  {
    Student s1;  // first student
    Student s2;  // second student

    s1 = new Student();
    s1.setFirst(”Adeeb”);
    s1.setLast(”Jarrah”);
    s2 = new Student(”Heejoo”, ”Chun”);
    s2.printFullName();
  } // end main
} // end class StudentDriver
```

Figure A5.2 StudentDriver class, used with the Student class in Figures A5.1a and A5.1b

Javadoc with Tags

Chapter 8’s Section 8.2 and Appendix 5 describe a programming style that is optimized for code presentation in a textbook and for students writing relatively simple programs. Most of the suggestions there carry over to professional programming practice. However, in professional programming, you should provide interface documentation for your classes that looks like the documentation Oracle provides for its Java application programming interface (API) classes.

As described in Section 8.3, you can run the javadoc executable that comes with your Java Development Kit (JDK) and generate this documentation automatically. To run javadoc, at a command prompt, enter this command:

```
javadoc –d output-directory source-files
```

The -d *output-directory* option[1] causes the output to go to another directory. If you omit this -d option symbol, by default, the output goes to the current directory, but that's not a good idea because javadoc creates many files that would clutter up the current directory. You can put documentation for more than one class in the same directory by putting spaces between the source-file names.

As indicated in Section 8.3, for a comment to appear on a javadoc web page, it must be located either immediately above a class heading (after import statements) or immediately above a method heading. In addition, it must be within a special (javadoc) comment block that starts with /** and ends with */. The opening and closing symbols may be on the same line as the enclosed comment, like this:

```
import java.util.Scanner;

/** This class handles processing of a student's name. */

public class Student_jd
{
```

Or the opening and closing symbols may be on separate lines, like this:

```
/**
Precondition: Each passed-in name must start with an uppercase
letter and all subsequent letters must be lowercase.
*/
public Student_jd(String first, String last)
{
```

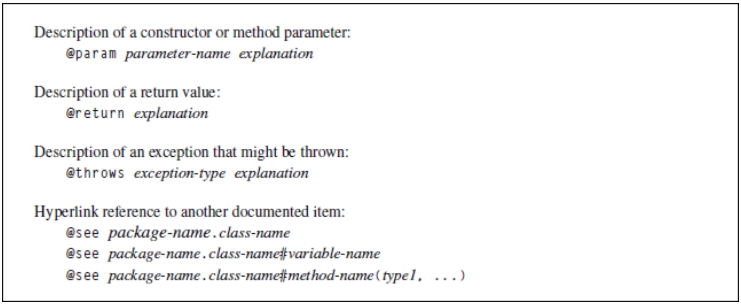
In both cases, it’s good style to leave a blank line above the javadoc comment. When the closing */ is on the same line as the commenting text, there should also be a blank line after the comment. When the closing */ is on a separate line, the following blank line is frequently omitted.

Within a /** . . . */ javadoc comment block, javadoc also recognizes several special tags, which enable it to extract other kinds of information. For a complete description, see:

https://www.oracle.com/java/

technologies/javase/javadoc-tool.html

Figure A6.1 contains an abbreviated list of javadoc tags.



Description of a constructor or method parameter:
`@param` *parameter-name explanation*

Description of a return value:
`@return` *explanation*

Description of an exception that might be thrown:
`@throws` *exception-type explanation*

Hyperlink reference to another documented item:
`@see` *package-name*.*class-name*
`@see` *package-name*.*class-name*#*variable-name*
`@see` *package-name*.*class-name*#*method-name*(*type1*, ...)

Figure A6.1 Abbreviated list of javadoc tags

Page 976

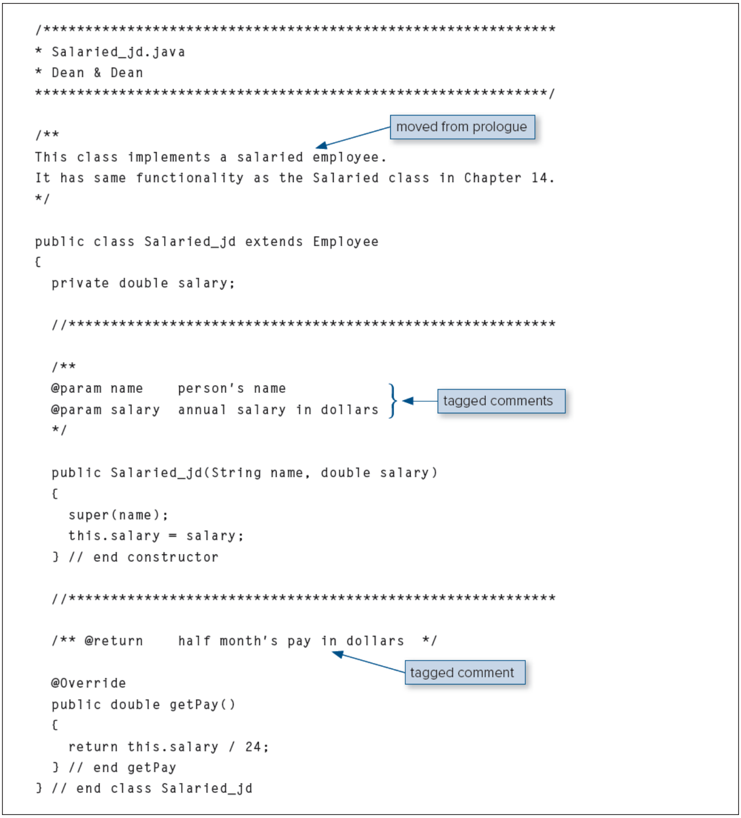
The most important tags are the @param and the @return tags. Figure A6.2 shows a class like that defined in Figure 14.11, but with its comments modified for javadoc. The functionality of this class is exactly the same as that defined in Figure 14.11. But this version enables several javadoc features. Notice how the general class description has been moved from the prologue into a separate javadoc comment block immediately above the class heading. In the javadoc comment block above the constructor there are two tagged parameter descriptions. In the javadoc comment block above the method there is a tagged return value description. The @Override is not a javadoc tag. It's a compiler tag.

Suppose the current directory contains source code for the Employee class copied from Figure 14.10, and it also contains the source code for the Salaried_jd class shown in Figure A6.2. Then suppose that we open a command-prompt window and enter the following command:

```
javadoc -d docs Employee.java Salaried_jd.java
```

This creates interface documentation for both classes and outputs that combined documentation to the docs subdirectory. Figure A6.3a shows what you'll see if you open a web browser, navigate to the docs directory, click on index.html, and select Salaried_jd in the left panel under “All Classes.”

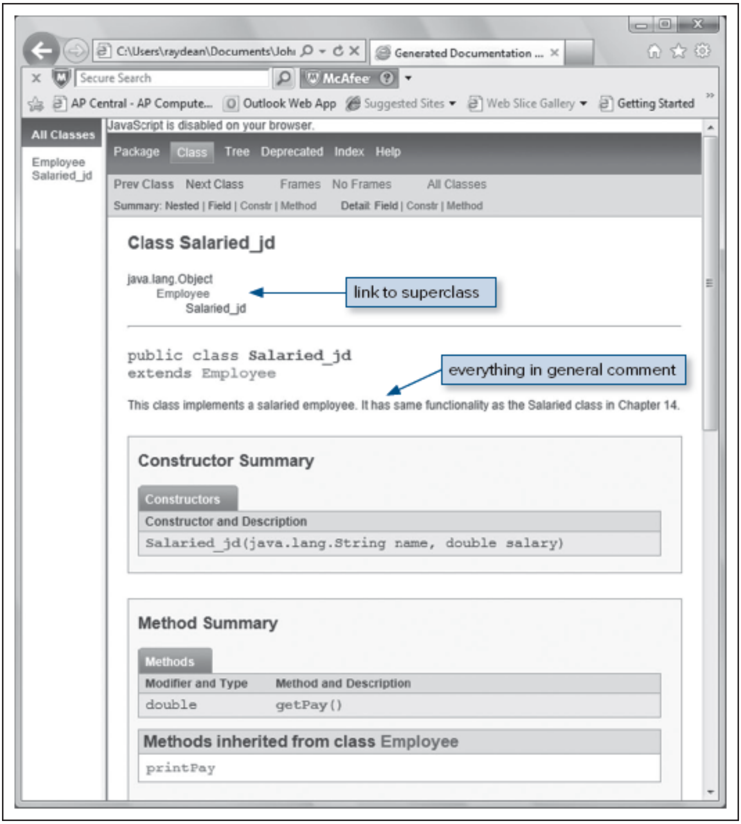
In Figure A6.3a’s right panel, near the top, you can see the documentation of Salaried_jd’s inheritance from Employee. In the Salaried_jd documentation, Employee is colored and underlined in several places. These are links, and if you click on any of them, the display switches immediately to the Employee class’s documentation. In Figure A6.2, our general comment had two sentences, and both of these sentences appear in the general comment in Figure A6.3a. Notice that the constructor and method summary blocks do not contain any comments. The @param and @return tags do not produce any summary-block output. If we had included text in the javadoc comment block above the constructor or method heading in Figure A6.2, only the first sentence of that text (the “summary” sentence) would appear in the corresponding summary block in Figure A6.3a.



```
/*************************************************************
* Salaried_jd.java
* Dean & Dean
*************************************************************/

/**
This class implements a salaried employee.
It has same functionality as the Salaried class in Chapter 14.
*/

public class Salaried_jd extends Employee
{
  private double salary;

  //**********************************************************

  /**
  @param name     person's name
  @param salary   annual salary in dollars
  */

  public Salaried_jd(String name, double salary)
  {
    super(name);
    this.salary = salary;
  } // end constructor

  //**********************************************************

  /** @return    half month's pay in dollars  */

  @Override
  public double getPay()
  {
    return this.salary / 24;
  } // end getPay
} // end class Salaried_jd
```

Figure A6.2 Modification of Figure 14.11's Salaried class with javadoc comments

Now suppose you use the scroll bar on the right to scroll down. This displays what you see in Figure A6.3b. Notice that the "Detail" blocks do display the tagged parameter and return information supplied in javadoc comment blocks above the constructor and method headings in Figure A6.2. If you had included text in a javadoc comment block preceding the constructor or method heading in Figure A6.2, all of this text would appear in the corresponding “Detail” block in Figure A6.3b. Finally, notice that javadoc also tells us that the getPay method defined in Salaried_id overrides a getPay method defined in Employee.


C:\Users\raydean\Documents\Joh
Generated Documentation ...
Secure Search
McAfee
AP Central - AP Compute...
Outlook Web App
Suggested Sites
Web Slice Gallery
Getting Started
All Classes
Employee
Salaried_jd
Package
Class
Tree
Deprecated
Index
Help
Prev Class
Next Class
Frames
No Frames
All Classes
Summary: Nested | Field | Constr | Method
Detail: Field | Constr | Method

JavaScript is disabled on your browser.

Class Salaried_jd

java.lang.Object
  Employee
    Salaried_jd

link to superclass

```
public class Salaried_jd
extends Employee
```

everything in general comment

This class implements a salaried employee. It has same functionality as the Salaried class in Chapter 14.

Constructor Summary

Constructors

Constructor and Description
`Salaried_jd(java.lang.String name, double salary)`

Method Summary

Methods

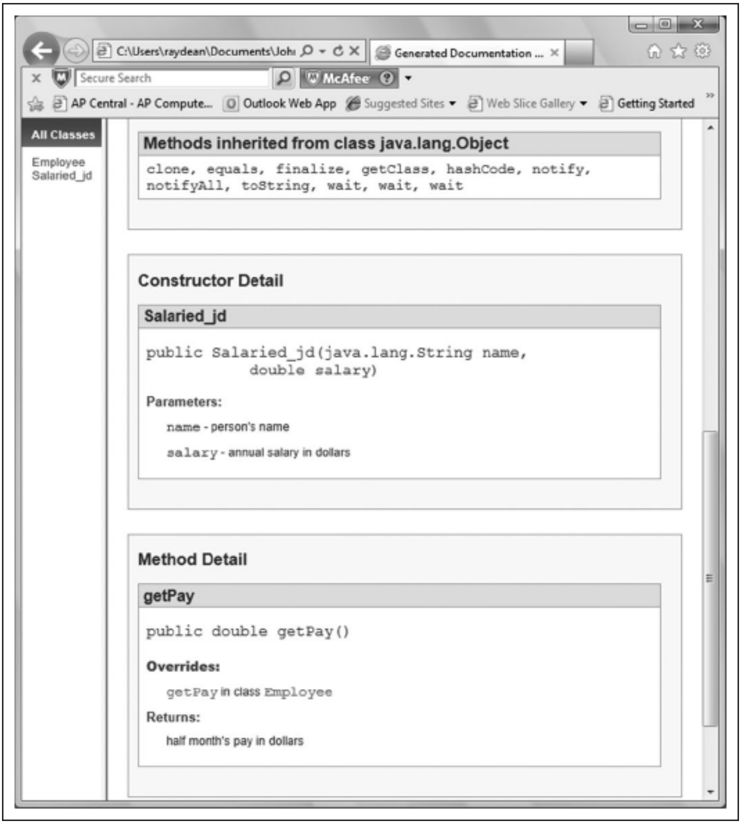
Modifier and Type	Method and Description
`double`	`getPay()`

Methods inherited from class Employee
`printPay`

Figure A6.3a javadoc output for the javadoc-commented Salaried class—part A

©Oracle/Java

Page 979


C:\Users\raydean\Documents\Joh
Generated Documentation ...
Secure Search
McAfee
AP Central - AP Compute...
Outlook Web App
Suggested Sites
Web Slice Gallery
Getting Started
All Classes
Employee
Salaried_jd

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify,
notifyAll, toString, wait, wait, wait
```

Constructor Detail

Salaried_jd

```
public Salaried_jd(java.lang.String name,
                   double salary)
```

Parameters:

`name` - person's name

`salary` - annual salary in dollars

Method Detail

getPay

```
public double getPay()
```

Overrides:

`getPay` in class `Employee`

Returns:

half month's pay in dollars

Figure A6.3b javadoc output for the javadoc-commented Salaried class—part B

©Oracle Corporation

UML Diagrams

The Unified Modeling Language (UML) is a descriptive language that helps program designers organize the subject matter of a prospective object-oriented program, and it provides high-level documentation of both structure and behavior. It's independent of any particular programming language, and it doesn't compile into an executable program. It's just an organizational tool. It was developed by the "Three Amigos"—Grady Booch, James Rumbaugh, and Ivar Jacobson, at Rational Software Corp., which is now part of IBM. Currently, it is maintained by the nonprofit Object Management Group (OMG) consortium.

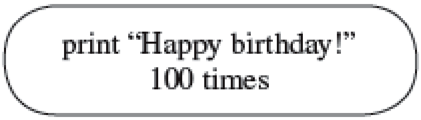
UML specifies many different kinds of visualizing diagrams.[1] In this appendix, we'll focus on just two of them—*activity diagrams* (which depict behavior) and *class diagrams* (which depict structure). When UML describes behavior, arrows point to what happens next. When UML describes structure, arrows point to what provides support, and this is opposite to the direction of "information flow." So in the following discussion, be prepared for a switch in arrow directionality as we move from activity diagrams to class diagrams.

UML Activity Diagrams

Activity diagrams are UML's version of the flowcharts we introduced in Chapter 2. They portray an algorithm's flow of control. Figure A7.1 shows an example of a UML activity diagram for the Happy Birthday algorithm presented as a flowchart in Figure 2.9.

The solid black circle is an *initial state,* and the black dot in a white circle is a *final state.* The oval boxes represent *action states* or *activities.* They contain informal descriptions of coherent actions. The arrows are *transitions.* The labels in square brackets next to some of the transitions are boolean conditions called *guards*—a particular transition occurs if and only if the adjacent guard value is true. The actions or activities shown in Figure A7.1 represent low-level or primitive operations.

At a higher level of scale, the activity described in a single oval could represent a whole set of actions. For example, you could use a single activity symbol to represent the whole looping operation shown in Figure A7.1 like this:



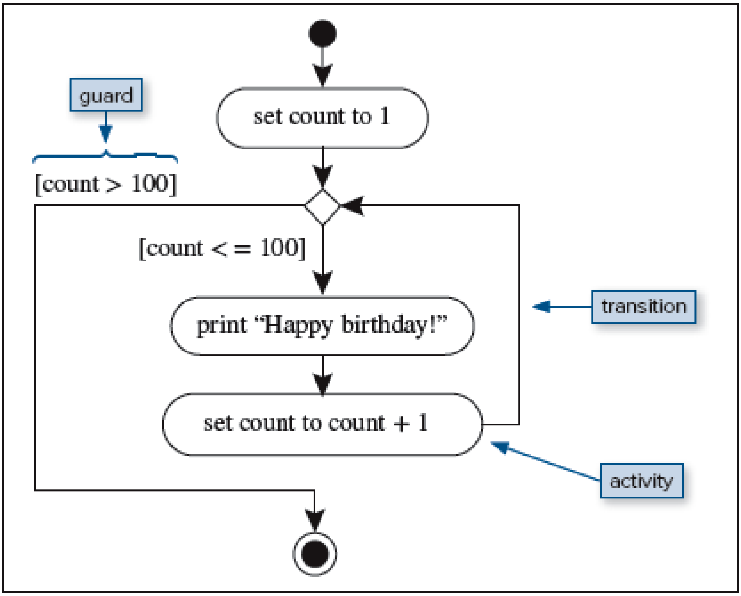

guard
set count to 1
[count > 100]
[count < = 100]
print "Happy birthday!"
transition
set count to count + 1
activity

Figure A7.1 UML activity diagram for the Happy Birthday algorithm in Figure 2.9

Or you could use a single activity symbol to represent all the actions performed by a complete method. An activity symbol is not supposed to represent code itself. It's supposed to represent the code's "activity." Thus, it's appropriate to repeat an activity symbol that represents a complete method when you call that method more than once.

When there is more than one class, and perhaps several objects, UML suggests that you organize the activities into columns, such that all the activities for any one class or object are in a single column dedicated to that class or object. UML calls these separate columns *swimlanes.* Vertical dashed lines separate adjacent swimlanes. At the top of the diagram, over appropriate swimlane(s), write the class name for the lane or lanes below. Precede each class name with a colon and put it into a separate rectangular box. When you mean to instantiate an object, write that object's name followed by a colon and its class name. Underline it and put it into a separate rectangular box located just after the activity that creates it.

Figure A7.2 shows the UML activity diagram for the Mouse2 program defined in Figures 6.13 and 6.14.

Notice how each activity (oval) is aligned under its own class and (if applicable) its own object. Activities for the lowest-level objects typically represent complete methods. Activities for higher-level objects typically represent code fragments. Solid black arrows represent control flow. They always go from one activity to another activity. Notice how the control flow moves continuously downward.

Dashed black arrows represent data flow associated with each activity. They go from an activity to an object, or from an object to an activity, but never from one activity to another activity. These dashed lines are often omitted to reduce clutter, but you can see how they help to show what the activities do. For example, notice how the dashed line from the "mickey : Mouse2" object to the "print mickey's attributes" activity helps explain what happens and allows us to suppress the two “get” method calls embedded in the print statement:

```
System.out.printf("Age = %d, weight = %.3f\n",
  mickey.getAge(), mickey.getWeight());
```

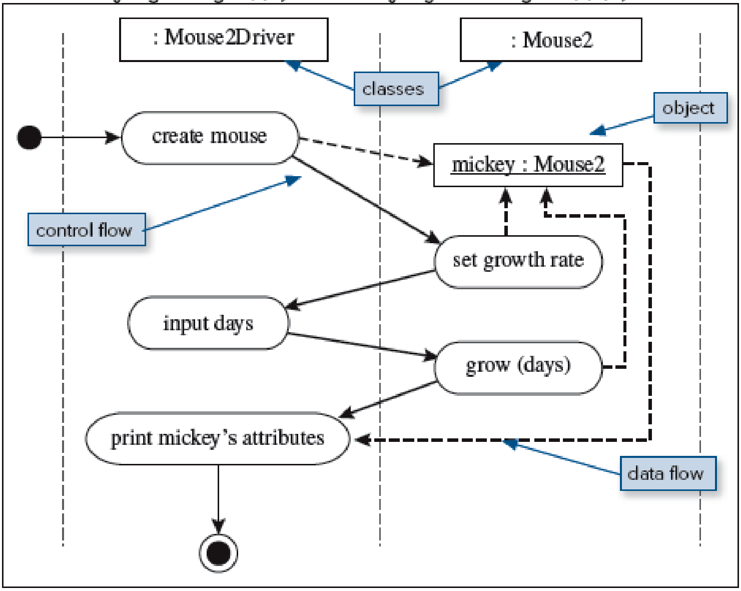

: Mouse2Driver
: Mouse2
classes
object
create mouse
mickey : Mouse2
control flow
set growth rate
input days
grow (days)
print mickey's attributes
data flow

Figure A7.2 UML activity diagram for the Mouse2 program in Figures 6.13 and 6.14.

Ovals are activities. Rectangles are classes or objects—objects are underlined. Dashed gray vertical lines separate adjacent *swimlanes*, with one lane for each class or object. Solid black arrows represent control flow. Dashed black arrows represent data flow.

The introduction of constructors in Chapter 7 makes it possible to include the “set growth rate” activity within the “create mouse” activity. This would replace the top two swimlane-crossing transitions with a single transition from the “create mouse” activity to the “input days” activity in the same left-side swimlane. Minimizing swimlane crossings is a good design goal.

UML Class and Object Diagrams

Starting in Chapter 6, we gradually introduced you to various features of UML class diagrams. UML object diagrams are similar, except the title (object name followed by a colon followed by class name) is underlined —as in the UML activity diagram in Figure A7.2. An object block does not include a methods compartment, and only those variables of current interest should be listed in the attribute compartment. Object diagrams are -

context-dependent snapshots, with attribute values being current values rather than initial values. Class diagrams have more general application, and from now on, we'll restrict our attention to them.

We'll use a comprehensive example to summarize most of the features of UML class diagrams presented throughout the main part of the book. The example we'll use is the Payroll3 program described in Section 14.9. Figure A7.3 portrays a first-cut class diagram in which each class is represented by a simple, one-compartment rectangle that contains nothing more than the class name. The solid lines drawn between related classes are simple association lines. A simple (unadorned) association line implies bidirectional knowledge—the class at each end knows about the class at the other end. Thus a simple line says dependencies are mutual, but it says nothing else about the nature of the relationship between connected classes.

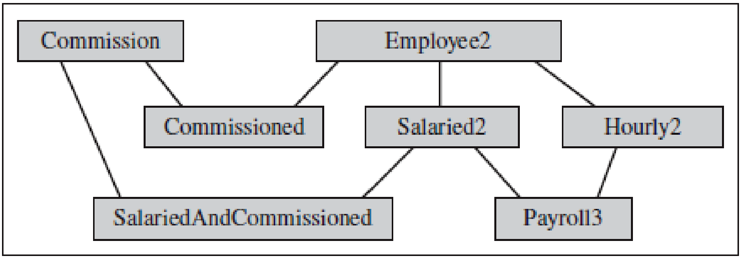

Commission
Employee2
Commissioned
Salaried2
Hourly2
SalariedAndCommissioned
Payroll3

Figure A7.3 First-cut UML class diagram for the Payroll3 program in Section 14.9

As you progress in your design thinking, you'll flesh out class descriptions, perhaps deciding to make some of the classes abstract or converting them to interfaces. In addition, you'll modify many of the association lines by adding special symbols that describe particular types of relationships. In addition, you might add barbed arrowheads to convert associations from bidirectional to unidirectional and make dependencies go just one way. Unidirectional dependencies are preferable to bidirectional dependencies because they simplify software management—software changes to one class are less likely to require changes to other classes.

Figure A7.4 contains a fleshed-out and modified version of the first-cut UML class diagram in Figure A7.3. Notice that we italicize the Commission and Employee2 class names. That means they have at least one abstract method and cannot be instantiated. We also italicize all the abstract methods they contain. Next, look at the hollow arrowheads, which indicate inheritance. Inheritance arrowheads on solid lines indicate extension of a class. Inheritance arrowheads on dashed lines indicate implementation of an interface. The arrowheads point in the direction of generalization—toward the more general entity. The more specific entities know about the more general entities and depend on them. Because of this dependency, changes to ancestor classes or interfaces can force changes to descendant or implementing classes. On the other hand, since an ancestral class or interface does not know about its descendants, changes in descendants or implementations never force changes in ancestors or interfaces. Inheritance is automatically a unidirectional association.

Now look at the composition indicators.2 We chose to show them as (solid diamond) compositions rather than (hollow diamond) aggregations because the class that instantiates the components (Payroll3) inserts anonymous components into its containing array. All the composition lines have multiplicities. These indicate that there is always exactly one payroll and there could be any number of employees of any of the four types. Because Hourly2, Salaried2, Commissioned, and SalariedAndCommissioned all descend from the Employee2 class, we can put instances of all four of these classes into a common Employees2 array, as we do in the Payroll3 class definition in Figure 14.17.

Finally, look at the barbed arrowheads we have added to the composition association lines. As we said, all association lines are bidirectional by default, and one design objective is to convert bidirectional associations into unidirectional associations. The barbed arrowheads on the non-diamond ends of the four composition lines do that. They say the composition's components have no knowledge of their container. That's appropriate in this case because this container is just a driver, and many drivers are ephemeral—here today and gone tomorrow.

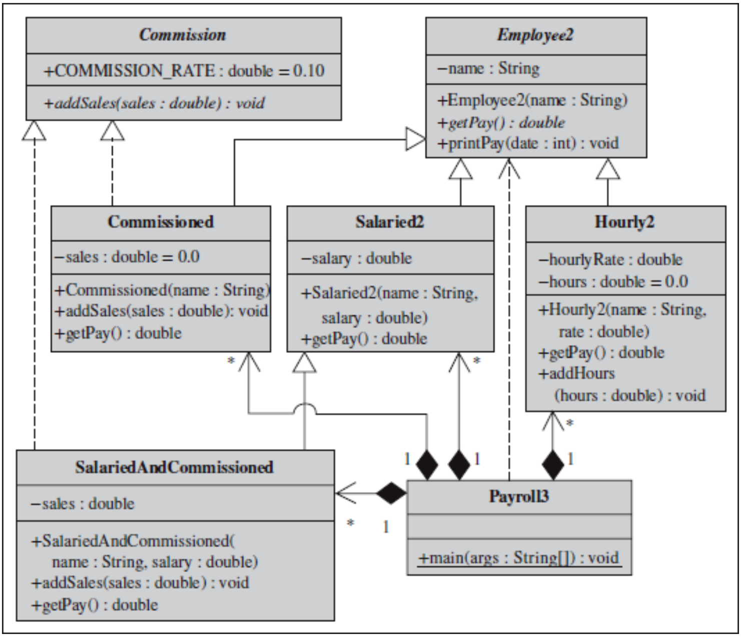

Commission
+COMMISSION_RATE : double = 0.10
+addSales(sales : double) : void
Employee2
−name : String
+Employee2(name : String)
+getPay() : double
+printPay(date : int) : void
Commissioned
−sales : double = 0.0
+Commissioned(name : String)
+addSales(sales : double): void
+getPay() : double
Salaried2
−salary : double
+Salaried2(name : String,
salary : double)
+getPay() : double
Hourly2
−hourlyRate : double
−hours : double = 0.0
+Hourly2(name : String,
rate : double)
+getPay() : double
+addHours
(hours : double) : void
*
*
*
1
1
1
SalariedAndCommissioned
−sales : double
+SalariedAndCommissioned(
name : String, salary : double)
+addSales(sales : double) : void
+getPay() : double
*
1
Payroll3
+main(args : String[]) : void

Figure A7.4 UML class diagram for the Payroll3 program

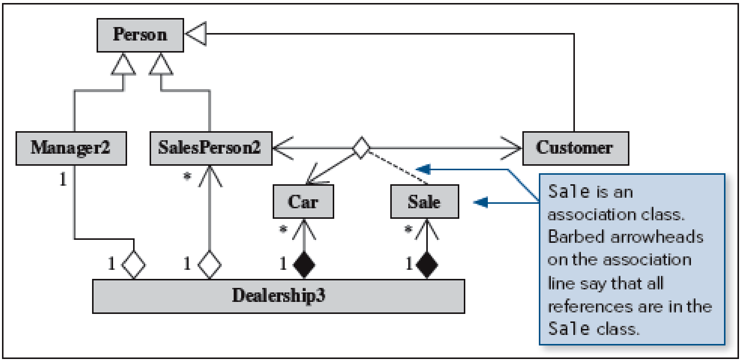
This shows inheritance from classes and implementation of an interface. It also shows composition. Because every association line in this figure has some kind of arrowhead, all its associations are unidirectional. The dashed association between Payroll3 and Employee2 is a simple dependence. That means the Employee2 type appears in the declaration of a parameter or local variable somewhere in Payroll3 ’s code.

Figure A7.4 also includes a dashed association line with a barbed arrow that points to the abstract class, Employee2. This acknowledges that main’s local variable, employees, depends on the class, Employee2, because the type of its elements is Employees2. The barbed arrow at the Employees2 end of this dashed association line indicates that the association is unidirectional. Payroll3 knows about Employees2, but Employees2 does not know about Payroll3. Thus, changes to Employees2 might require changes to Payroll3, but changes to Payroll3 never require changes to Employees2. UML uses dashed association lines for parameter and local variable dependencies, and it uses solid association lines for instance and static variable dependencies.

As described in Section 13.12, UML

also uses dashed association lines to connect an association class to an association between or among other classes. Figure 13.25 shows a solid association line connecting the three classes: SalesPerson2, Customer, and Car. Although we did not discuss this detail in Chapter 13, the fact that this association line is solid and has no barbed arrowheads at its ends suggests that each of these three classes has instance variables that refer to particular instances of the other two classes.

The association class called Sale makes such additional references unnecessary because the Sale class can hold all these references itself, in one place. Thus, this extra association class reduces the number of reference variables. More importantly, it eliminates the need to alter the definition of the SalesPerson2 and Car classes when we add a Customer class and a Sale association to the program. To reflect the fact that the SalesPerson2, Car, and Customer classes do not need any references to instances of other classes in the common association, we put barbed arrowheads on the three ends of the association line that connects them. This changes Figure 13.25 to what appears in Figure A7.5.[3]



Sale is an association class. Barbed arrowheads on the association line say that all references are in the Sale class.

Figure A7.5 Improved version of the class diagram in Figure 13.25.

An arrowhead on an association line means the adjacent class does not have references to the other classes in that association.

Notice that Figure A7.5 also includes a composition association between Dealership3 and Sale. The barbed arrowheads at the Sale and Car ends of their respective composition lines and at the SalesPerson2 end of its aggregation line say that Dealership3 depends on these other classes. In other words, Dealership3 has references to instances of the Sale, Car, and SalesPerson2 classes, but not vice versa. In contrast, the aggregation association between Dealership3 and Manager2 does not have any arrowheads. This says each has a reference to the other.

Number Systems and Conversions Between Them

Probably because we have ten fingers, ordinary humans use a base-ten number system called *decimal*. In the decimal system, the smallest component is a *digit*. Each digit has one of ten distinct values, 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. Computer programmers also use two other number systems, *binary* and *hexadecimal*.

In Chapter 1 Section 1.2, we provided a brief introduction to the base-two number system, called *binary*. In the binary system, the smallest component is a *bit*. A bit has one of only two distinct values, 0 and 1. In Chapter 12 Section 12.2, we used bits to describe the amount of memory required to hold different types of Java numbers.

In Section 12.15, we used binary numbers to describe the base-sixteen number system, called *hexadecimal*. In the hexadecimal system, the smallest component is a *hexit*. A hexit has one of sixteen distinct values, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. In hexadecimal, A, B, C, D, E, and F correspond to decimal 10, 11, 12, 13, 14, and 15, respectively. Instead of the uppercase letters *A, B, C, D, E,* and *F,* Java allows us to substitute the lowercase letters *a, b, c, d, e,* and *f*. Computers typically use hexadecimal for memory addressing, and as described in Sections 12.16 and 17.15 and Appendix 1, they also use hexadecimal to represent characters and colors.

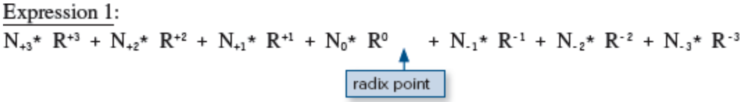
Sometimes people use a base-eight number system, which is called *octal*. In principle, the base can be any positive integer.

To represent a number greater than its base, a number system uses multiple components. It puts each component in a different position, and it assigns each position a distinctive weight. In any one number system, the weights are distinct integer powers of that system's base, with the powers in descending order.[1] For example, when we write a decimal number like 1609.34, what we actually mean is

$$1*10^{+3} + 6*10^{+2} + 0*10^{+1} + 9*10^{0} \underset{\substack{\uparrow \\ \boxed{\text{decimal point}}}}{} + 3*10^{-1} + 4*10^{-2}$$

The decimal point separates the integer part of the number on the left from the fraction part on the right, and the decimal point's location determines the weight in each component position.

It's easy to generalize this decimal example. For the more general case, let's rewrite this number using N_{+3}, N_{+2}, N_{+1}, N_0, N_{-1}, N_{-2}, and N_{-3} for the number’s components, like 1, 6, 0, 9, 3, 4, and a trailing 0. The mathematical term for the base is *radix*, so we use R for the base value.

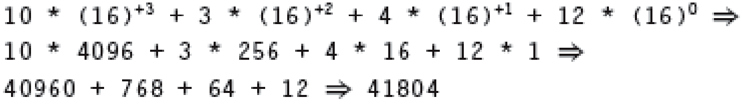


Expression 1:

$N_{+3}* R^{+3} + N_{+2}* R^{+2} + N_{+1}* R^{+1} + N_0* R^0 \quad + N_{-1}* R^{-1} + N_{-2}* R^{-2} + N_{-3}* R^{-3}$

“In” Conversion—from Another Base to Decimal

Now, suppose we have a memory location in hexadecimal, like A34C. What is that in decimal? Because it is hexadecimal, R is 16. Because it has no radix point, it is an integer. Recognizing that hexadecimal A is decimal 10 and hexadecimal C is decimal 12, N_{+3} is 10, N_{+2} is 3, N_{+1} is 4, and N_0 is 12. So with A34C in Expression 1, the evaluation looks like this:



$10 * (16)^{+3} + 3 * (16)^{+2} + 4 * (16)^{+1} + 12 * (16)^0 \Rightarrow$

$10 * 4096 + 3 * 256 + 4 * 16 + 12 * 1 \Rightarrow$

$40960 + 768 + 64 + 12 \Rightarrow 41804$

Therefore, A34C$_{\text{hexadecimal}}$ is the same as 41804$_{\text{decimal}}$.

This evaluation requires three additions and six multiplications (assuming we drop the unnecessary final multiplication by unity). Expression 1 is a polynomial. When we use this algorithm to evaluate a polynomial, the number of multiplications increases as the square of the number of terms. If we care about efficiency, either in a computer or in a hand calculation, wherever we evaluate a polynomial, we should not do what we have just done. There is a better way—namely, to group things using parentheses, like this:

Expression 2:

$$((N_{+3}*R + N_{+2})*R + N_{+1})*R + N_0 \quad \underset{\substack{\uparrow \\ \text{radix point}}}{} \quad + ((N_{-3}/R + N_{-2})/R + N_{-1})/R$$

Notice how similar the fractional part of Expression 2 is to the integer part. If you know the integer part and want the fractional part, just copy the integer part without N$_0$, and then replace each plus subscript with a minus subscript and replace each multiplication with a division. Expression 2 tells how to convert integers and fractions from another base to decimal with best efficiency. You'll see shortly that this same expression also tells how to convert integers and fractions from decimal to another base. For conversions "in" from another base, calculations move from the outer components N_{+3} and N_{-3} "in" toward the radix point. For conversions "out" to another base, you'll see shortly that calculations move from the radix point "out" to the outer components, N_{+3} and N_{-3}. Whenever you write this expression, remember that R is always the other base. It's never ten.

Now let's repeat our previous calculation of the decimal equivalent of a hexadecimal integer, using Expression 2 instead of Expression 1.

With the same hexadecimal integer, A34C, in Expression 2, the evaluation looks like this:

$$((10 * 16 + 3) * 16 + 4) * 16 + 12 \Rightarrow$$

$$(163 * 16 + 4) * 16 + 12 \Rightarrow 41804$$

Of course, we get the same result as before. But this time, the number of multiplications increases linearly with the number of terms, so there are only three multiplications instead of six. Using a typical hand calculator, we just enter the numbers and operators in the order in which they appear, and update the evaluation at each closing parenthesis.

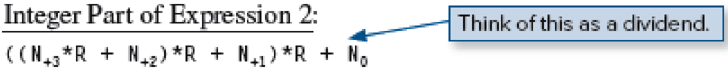
Converting the fractional part of a fractional hexadecimal number is also easier with Expression 2 than with Expression 1. Suppose we have the hexadecimal fraction, 0.5B7. What is that in decimal? Because it is hexadecimal, R is 16. Because it does have a radix point, it is a fraction. Recognizing that hexadecimal B is decimal 11, N_{-3} is 7, N_{-2} is 11, and N_{-1} is 5. So with 0.5B7 in expression 2, the evaluation looks like this:

$$((7 / 16 + 11) / 16 + 5) / 16 \Rightarrow$$
$$(11.4375 / 16 + 5) / 16 \Rightarrow$$
$$5.71484375 / 16 \Rightarrow 0.357177734$$

Therefore, $0.5B7_{\text{hexadecimal}}$ is the same as $0.357177734_{\text{decimal}}$.

“Out” Conversion—from Decimal to Another Base

Expression 2 also makes it easy to remember how to perform conversion in the other direction—from a decimal number to the same number in another base. To see how this works, let’s start with an integer or the integer part of a floating-point number. This is what is to the left of the radix point in Expression 2:



Integer Part of Expression 2:

$$((N_{+3}*R + N_{+2})*R + N_{+1})*R + N_0$$

Think of this as a dividend.

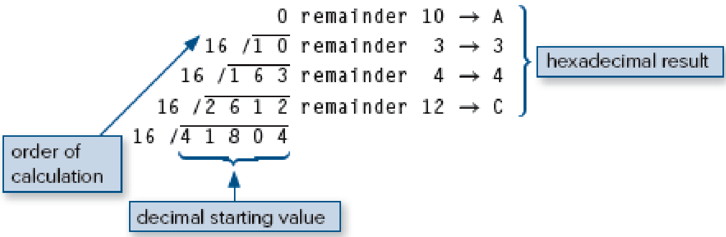
Think of this expression as the dividend in a division problem. If we divide this dividend by the base, R, the quotient is

$$(N_{+3}*R + N_{+2})*R + N_{+1}$$

And the remainder is N_0. This remainder is the least-significant integer component in the other base.

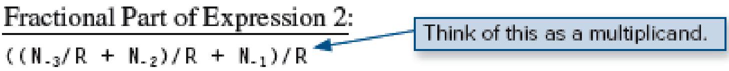
Now, consider this quotient to be a dividend and divide it by the base, R. This time, the remainder is N_{+1}. Do this again, and the remainder is N_{+2}. Continue until the quotient is zero. The remainders generated by this algorithm are the other base's integer components, from least significant to most significant.

If you're doing it by hand, you need to store intermediate results, and if you're like us, even if your hand calculator has storage, you'll want to use pencil and paper to keep track of your progress. Suppose you want to reverse our previous conversion that led to $41804_{decimal}$. The picture below illustrates what you might write on the paper as you proceed. The picture shows a sequence of divisions by the new base value, 16, starting at the bottom and moving up. The result is the reverse of the remainder sequence. When you finish, the converted number appears vertically on the right, read from top to bottom.



0 remainder 10 → A
16 /1 0 remainder 3 → 3
16 /1 6 3 remainder 4 → 4
16 /2 6 1 2 remainder 12 → C
16 /4 1 8 0 4
hexadecimal result
order of calculation
decimal starting value

Now consider the fractional part of a floating-point number by looking at what's to the right of the radix point in Expression 2:



Fractional Part of Expression 2:
((N-3/R + N-2)/R + N-1)/R
Think of this as a multiplicand.

Think of this expression as the

multiplicand in a multiplication problem. Multiply this multiplicand by the base, R, to obtain the floating-point quotient:

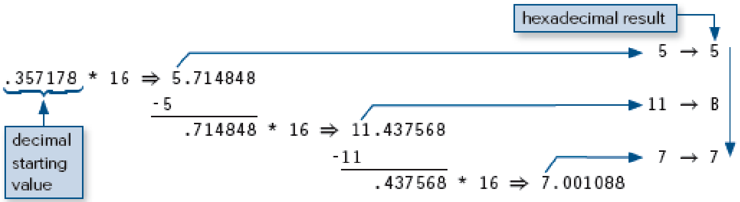
$$(N_{-3}/R + N_{-2})/R + N_{-1}$$

The last term, N_{-1}, is the integer part of this floating-point quotient. It is the most significant component of the original fraction. Now look at the fractional part of this floating-point quotient:

$$(N_{-3}/R + N_{-2})/R$$

Consider this to be another multiplicand. Multiply it by the base, R, to obtain another fraction followed by the integer N_{-2}, which is the second most significant component of the original fraction. Continue until the fractional part equals zero or you have as much precision as you want.

To illustrate this process, we'll start with a rounded version of the decimal fraction we generated before, 0.357178, and make the conversion in the other direction, from decimal to hexadecimal. With a simple hand calculator, this conversion is easier than the integer conversion. It doesn't require any storage—just use pencil and paper to write down the resulting components as they appear. The picture below shows how the calculation proceeds.


hexadecimal result
.357178 * 16 ⇒ 5.714848
-5
.714848 * 16 ⇒ 11.437568
-11
.437568 * 16 ⇒ 7.001088
5 → 5
11 → B
7 → 7
decimal starting value

---

[1] Alternatively, you do not need to unzip the downloaded core Java JDK. In this alternative your new Path entry should be the path to the unzipped file with .zip extension.

[2] You can see all command options by entering javac -help or java -help.

[3] If we wanted to distribute this

module, instead of using just ipwj.ch01m we would use a longer institutionalized name like com.mheducation.ipwj.ch01m.

[4] In our examples we put @\fx.txt before the destination and other directives, but it could go anywhere between the initial command and the final filename.

[5] If you try to execute from any directory below the module directory, like this:

```
E:\mods\ipwj.ch01m\ipwj\ch01>java Hello
```

the compiler complains like this:

```
Error: Could not find or load main class Hello
Caused by: java.lang.NoClassDefFoundError: ipwj\ch01\Hello (wrong name: Hello)
```

[6] In a non-GUI application you can execute with slashes in the package

hierarchy, but if you use any slashes in a GUI application, the compiler complains like this:

```
Missing JavaFX application class ipwj.ch01.gui\TitleHello
```

[7] It's also possible to execute in a directory above the module directory, but the dolphinsC.jpg resource file must be in the execution directory.

[1] For other options and arguments, enter javadoc -help.

[1] The full UML specification at https://omg.org/spec/UML/2.5.1/PDF is almost 800 pages long. For a more readable introduction to UML 2.0, go to https://www.omg.org/news/meetings/workshops/MDA-SOA-WS_Manual/00-T4_Matthews.pdf and see, for example, the 21-page Activity Diagram section.

[2] In Figure A7.4, notice how the association line between Payroll3 and Commissioned arcs over the

association line between SalariedAndCommissioned and Salaried2. This UML detail helps distinguish a cross-over from a junction.

[3] Notice the small diamond at the intersection of the Sale association lines. This UML detail helps distinguish a junction from a cross-over.

[1] When the highest powers come first, it's called a *big endian*. Because mathematical operations like addition, subtraction, and multiplication start with the least-significant elements, computer hardware sometimes arranges bits with the lowest powers first. That's called *little endian*.

Writing Methods in a Non-Object-Oriented Environment

Objectives

- Write programs with several methods in the same class.
- Understand scope, persistence,

and use of the `static` modifier.

- Understand different ways to terminate method execution.
- Understand a method’s type.
- Learn how to pass data into and out of methods.
- Understand the behavior of arguments and parameters.
- Learn how to trace execution through method calls and returns.
- Learn how to overload a method name.
- Use `private` to hide subordinate helper methods.

Outline

S6.1 Introduction

S6.2 Methods Overview

S6.3 Example

UsernamePasswordEntry Program

S6.4 Local Variables

S6.5 Returning a Value

S6.6 Method Calls Without Data Transfer

S6.7 Argument Passing

S6.8 Tracing a Program with More than One Method

S6.9 Overloaded Methods

S6.10 Helper Methods

S6.1 Introduction

Chapter 5 served as a bridge from basic programming language constructs (variables, assignments, operators, `if` statements, loops, etc.) to modular concepts. There we focused primarily on one important aspect of modularization—calling prebuilt methods from your `main` method. The Interlude between Chapter 5 and Chapter 6 showed you how to do more than just call prebuilt methods. You learned how to write your own methods that can be called from `main`. The purpose of writing methods in addition to the `main` method is to partition your programs into separate modules, where each module solves a particular subtask. That sort of partitioning makes large programs easier to work with. Making large programs easier to work with is very important because today's computers use lots of very large programs!

In this chapter, we start with an overview of basic method concepts. We then lead you through the design and implementation of a simple program that implements two methods—`main` and a method that's called from `main`. This complements what you learned in the Interlude. It provides another example of the use of method *parameters*, which receive data from a method call. It also illustrates use of a separate set of *local variables*, which store data that's hidden within a particular method. You'll learn how to adapt the previously described tracing technique to a program with multiple methods. You'll learn how to use the same name for multiple methods that perform only slightly different tasks. And finally you'll learn how to delegate subtasks to hidden subordinate methods.

S6.2 Methods Overview

A method is an isolated chunk of code that accomplishes one task. Often, the task is small, like calculating the area of something. But the task can also be big, like playing a game of tic-tac-toe. To implement a big task, write the task's `main` method first and postpone the implementation of subtask details by calling other methods to perform the subtasks.

Here are some of the benefits of partitioning a program into separate methods:

- Readability—If you move some of the details from a big method into one or more smaller methods, that will make the original method smaller. And as long as the method partitioning makes sense, having smaller methods will lead to programs that are more readable (i.e., easier to understand).
- Reusability—If a particular subtask needs to be performed at more than one place within a large method and you implement the subtask with its own method, that's an example of code re-use because you're using the same code in more than one place. Code

re-use is beneficial because it means that you don't have to re-think or re-enter a task's code more than one time.

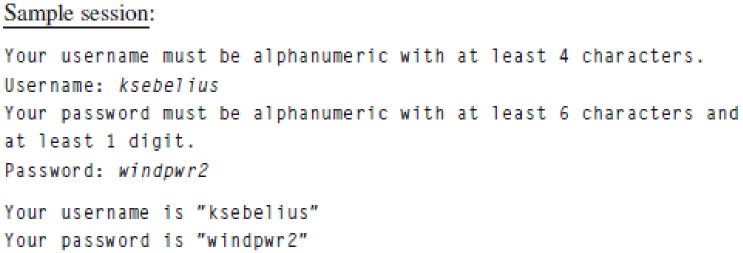
- Maintainability—If a particular subtask needs to be performed at more than one place within a program and you implement the subtask with its own method, that can make the program easier to maintain. Suppose the subtask's method needs to be fixed or enhanced. You will have to make the change in just one place. On the other hand, if you implement the subtask with redundant code (i.e., you insert the complete code at multiple places within your program) and the code needs to be fixed or enhanced, you will have to make the change in multiple places. That can lead to inconsistencies and errors.
- Encapsulation—A method can

define local variables that hide the method’s internal data from the outside world. Such data hiding is beneficial because it reduces the danger of having activity in one part of a program accidentally interfere with data in another part of the program.

S6.3 Example UsernamePasswordEntry Program

Suppose you want a program to prompt the user for two inputs—a username and a password. After each input, the program should check to see whether the user-entered data meets certain criteria. The input username must have at least 4 characters, and the characters must be either letters or numbers. The input password must have at least 6 characters, the characters must be either letters or numbers, and at least one of the characters must be a number. If the user-entered data is invalid, the program should generate an error message and then re-prompt the user to enter valid input. The program should repeat that process until the user-entered data is valid.

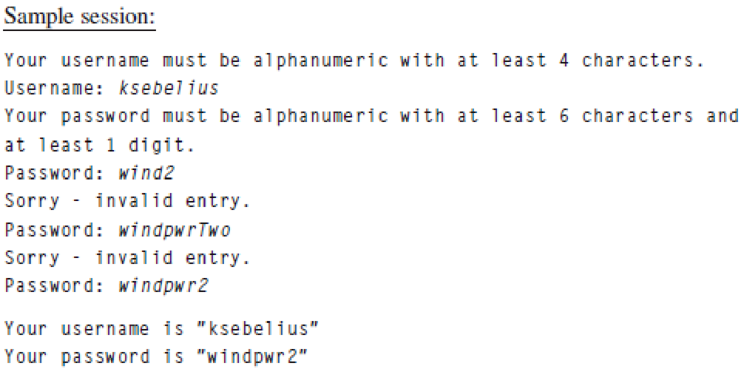
If the user's input conforms to the required constraints, the program should produce output like this:



Sample session:

```
Your username must be alphanumeric with at least 4 characters.
Username: ksebelius
Your password must be alphanumeric with at least 6 characters and
at least 1 digit.
Password: windpwr2

Your username is "ksebelius"
Your password is "windpwr2"
```

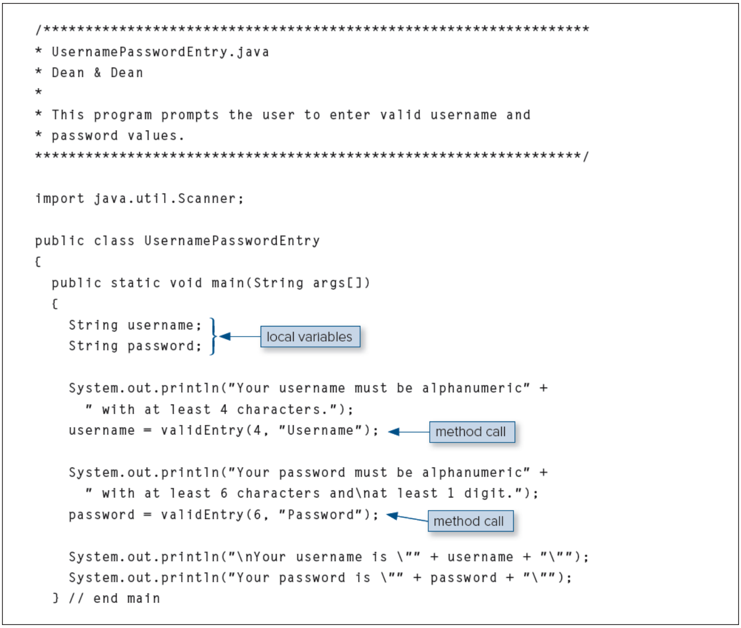
If the user enters a valid username and invalid passwords, the program should produce output like this:



Sample session:

```
Your username must be alphanumeric with at least 4 characters.
Username: ksebelius
Your password must be alphanumeric with at least 6 characters and
at least 1 digit.
Password: wind2
Sorry - invalid entry.
Password: windpwrTwo
Sorry - invalid entry.
Password: windpwr2

Your username is "ksebelius"
Your password is "windpwr2"
```

Figures S6.1a and S6.1b show the UsernamePasswordEntry program, which implements the functionality described above. The program's `main` method prints the first, third, and fourth lines of the above output and the last two lines of the above output. But it delegates other output, input, and input validation operations to a subordinate `validEntry` method. In particular, note how the program avoids code duplication by calling the same subordinate method, `validEntry` for username entry and password entry. In the first subordinate method call, the calling arguments specify the minimum number of characters, 4, and the prompting word, "Username." In the second subordinate method call, the calling arguments specify the minimum number of characters, 6, and the prompting word, "Password." These different arguments make the subordinate method behave in an appropriately different way for each of the two tasks.

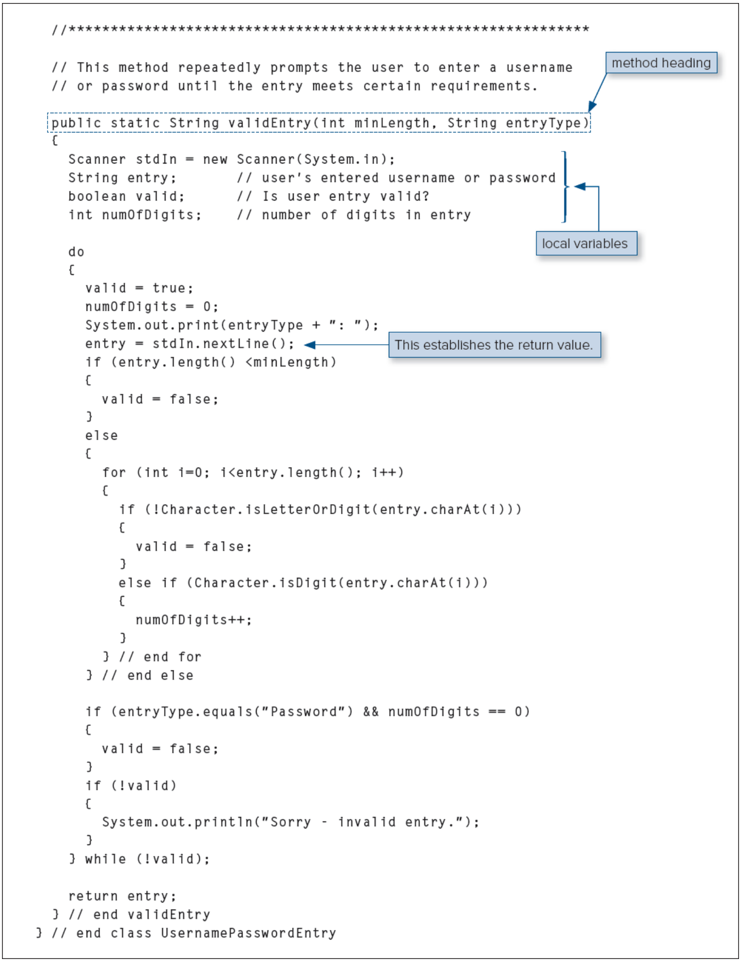


```
/*******************************************************************
* UsernamePasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter valid username and
* password values.
*******************************************************************/

import java.util.Scanner;

public class UsernamePasswordEntry
{
  public static void main(String args[])
  {
    String username;
    String password;

    System.out.println("Your username must be alphanumeric" +
      " with at least 4 characters.");
    username = validEntry(4, "Username");

    System.out.println("Your password must be alphanumeric" +
      " with at least 6 characters and\nat least 1 digit.");
    password = validEntry(6, "Password");

    System.out.println("\nYour username is \"" + username + "\"");
    System.out.println("Your password is \"" + password + "\"");
  } // end main
```

Figure S6.1a UsernamePasswordEntry program—part A

Figure S6.1b contains the remainder of the UsernamePassword Entry program—the subordinate `validEntry` method. The method prompts the user to enter a value. The first time the method is called, the prompt will be for the username. The second time the method is called, the prompt will be for the password. The user entry is then checked to see if it contains the minimum number of characters and if each of its characters is either a letter or a digit. For passwords, the method checks one more thing—whether the user entry contains at least one digit. If all those criteria are met, then the `validEntry` method returns the user entry back to the `main` method. Otherwise, the `validEntry` method warns the user and loops back for another user entry attempt.

Note the description at the top of Figure S6.1b. Proper coding conventions suggest that above each method, you should insert a blank line (the white space at the bottom of Figure S6.1a), a line of asterisks, another blank line, a description of the method, and another blank line. The blank lines and asterisks serve to separate different methods. The method description helps someone who’s reading your program to quickly get an idea of what’s going on.



```
  //*******************************************************************

  // This method repeatedly prompts the user to enter a username
  // or password until the entry meets certain requirements.

  public static String validEntry(int minLength, String entryType)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;       // user's entered username or password
    boolean valid;      // Is user entry valid?
    int numOfDigits;    // number of digits in entry

    do
    {
      valid = true;
      numOfDigits = 0;
      System.out.print(entryType + ”: ”);
      entry = stdIn.nextLine();
      if (entry.length() <minLength)
      {
        valid = false;
      }
      else
      {
        for (int i=0; i<entry.length(); i++)
        {
          if (!Character.isLetterOrDigit(entry.charAt(i)))
          {
            valid = false;
          }
          else if (Character.isDigit(entry.charAt(i)))
          {
            numOfDigits++;
          }
        } // end for
      } // end else

      if (entryType.equals(”Password”) && numOfDigits == 0)
      {
        valid = false;
      }
      if (!valid)
      {
        System.out.println(”Sorry - invalid entry.”);
      }
    } while (!valid);

    return entry;
  } // end validEntry
} // end class UsernamePasswordEntry
```

method heading

local variables

This establishes the return value.

Figure S6.1b UsernamePasswordEntry program—part B

Method Heading

The method itself consists of a *heading* and a *body*. Note `validEntry`'s heading in Figure S6.1b. Also note `validEntry`'s body, which consists of everything below the heading down to the method's closing brace:

```
  } // end validEntry
```

Let's now examine the `validEntry` method heading in depth. The `public` modifier establishes the method's accessibility. Accessibility can be either `public` or `private`. Given the standard definitions of the words "public" and "private," you can probably surmise that a `public` method is easier to access than a `private` method. If you declare a method to be `public`, the method can be accessed from anywhere (from within the method's class, and also from outside the method's class). When you want a method to perform a local task only, you should declare it to be `private`. We'll discuss `private` methods in more detail in a later section.

In the UserPasswordEntry program above, the `validEntry` method performs a local task, so it could have been declared `private`. But we chose to make it `public` so that, as an option, it could also be accessed from the outside world—from a different class. In other words, since the `validEntry` method is `public`, you can use it like you use the `public` Java API methods described in Chapter 5.

Note the `static` modifier in `validEntry`'s heading. The `static` modifier establishes the method as a class method. As you might recall, class methods are associated with an entire class, not a particular instance of a class. And that makes class methods easier to work with. If we had omitted the `static` modifier in `validEntry`'s heading, then extra work would be necessary in order to use the method. Specifically, you'd have to instantiate a `UserPasswordEntry` object and use that object to call `validEntry`. You'll get to that sort of thing later on, when you jump into the deep end of the object-oriented-programming pool. Since `validEntry` is a class method, you can call it easily, as shown in this code fragment from Figure S6.1a:

```
  username = validEntry(4,
"Username");
```

You might recall that with class methods found in the `Math` class, you call them by prefacing the call with the method's class name, `Math`. For example:

```
  futureValue = presentValue * Math.pow((1 + interestRate), years);
```

In Figure S6.1a's `main` method, you could preface `validEntry`'s method call with `validEntry`'s class name, `UserPasswordEntry`, but it's unnecessary. Why? Because `main` and `validEntry` are in the same class.

Note the third word in `validEntry`'s heading—`String`. `String` is the method's return type, and it specifies the type of data that the method will return. We'll discuss the return operation in a later section.

Finally, note `minLength` and `entryType` inside the parentheses in `validEntry`'s method heading. Those are *parameters*. Each parameter definition consists of the parameter's type followed by the parameter's name. Parameters are in charge of receiving and storing argument values that come from method calls. Here's the first call to `validEntry`, copied from Figure S6.1a:

```
username = validEntry(4, "Username");
```

The argument `4` gets passed to the parameter `minLength`, and the argument `"Username"` gets passed to the parameter `entryType`.

S6.4 Local Variables

A *local variable* is a variable that’s declared and used “locally” inside a method or in a `for` loop header. As you perhaps now realize, all the variables we defined in prior chapters were local variables. They were all declared within `main` methods, so they were all local variables within the `main` method, i.e., they were accessible only from within the `main` method. We didn’t bother to explain the term “local variable” until now because we did not create any other methods besides `main`, and the idea of a variable being local to `main` wouldn't have made much sense.

Scope

A local variable has *local scope*—it can be used only from the point at which the variable is declared to the end of the variable's *block*. A variable's block is established by the closest pair of braces that enclose the variable's declaration. Most of the time, you should declare a method's local variables at the top of the method's body. The scope of such variables is then the entire body of the method.

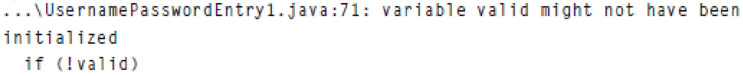
`for` loop index variables are local variables, but they are special. Their scope rule is slightly different from what is described above. As you know from Chapter 4, you should normally declare a `for` loop's index variable within the `for` loop's header. The scope of such a variable is the `for` loop's header plus the `for` loop's body.

Method parameters are usually not referred to as "local variables," but they are very similar to local variables in that they are declared and used "locally" inside a method. As with local variables, the scope of a method's parameters is limited to within the body of that method.

The UsernamePasswordEntry program in Figures S6.1a and S6.1b has two variables that are local to the `main` method (`username` and `password`), four variables that are local to the `validEntry` method (`entry`, `valid`, `numOfDigits`, and `stdIn`), and two parameters that are local to the `validEntry` method (`minLength` and `entryType`).

Frequently, local variables are not initialized when they are declared. That's the case for all of the local variables in the Figures S6.1a and S6.1b, except for `stdIn`. The initial value of any local variable that is not specifically initialized is *garbage*. Garbage means that the variable's value is unknown—it's whatever just happens to be in memory at the time the variable is created. All local variables must be given non-garbage values before they can be accessed.

If a program's logic is such that there might possibly be an attempt to access a variable that contains garbage, the compiler will generate a compilation error. For example, what might happen if the `valid = true;` statement were removed from the top of the `do` loop header in Figure S6.1b? It's possible that when the program runs, the user might enter a valid value right off the bat.1 In that case, none of the subsequent `valid = false;` assignment statements would get executed and `valid` would retain its original garbage value. Then when the `if (!valid)` line is reached, there would be an attempt to access a variable that contains garbage. The compiler is able to recognize this potential problem. So if the `valid = true;` statement were removed from the top of the `do` loop header in Figure S6.1b, the compiler would generate this error message:



```
...\UsernamePasswordEntry1.java:71: variable valid might not have been
initialized
  if (!valid)
```

Local Variable Persistence

OK, let's say you do initialize a local variable. How long will it *persist*? A local variable (or parameter) persists only within its scope and only for the current duration of the method (or `for` loop) in which it is defined. The next time the method (or `for` loop) is called, the local variable's value resets to garbage or the value given to it by initialization.

S6.5 Returning a Value

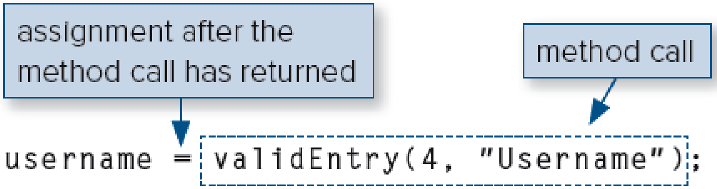
In this section, we discuss in depth the process of returning a value from a method. Since the primary purpose of Figure S6.1b's `validEntry` method is to return a validated user entry, we'll use that method as the vehicle for our discussion. Recall that the `validEntry` method heading, copied below for your convenience, includes a `String` return type:

```
public static String validEntry(int minLength, String entryType)
```

The `String` return type must match the type of the value that's returned from the method. That is indeed the case because near the top of the `validEntry` method, the local variable `entry` is declared to be a `String`. And at the bottom of the `validEntry` method, `entry`'s value is returned by virtue of this statement:

```
return entry;
```

The `return` statement passes a copy of the `entry` value back to the place where the method was called. Figure S6.1a's `main` method calls the `validEntry` method two times. Here's the first `validEntry` method call:



```
username = validEntry(4, "Username");
```

After the `validEntry` method has completed its execution, the Java Virtual Machine (JVM) returns the validated username and effectively substitutes that username for the `validEntry(4, "username")` method call in the `main` method. To perform a mental trace, imagine that the method call is overlaid by the returned value. So if the user enters "ksebelius" for a username, then "ksebelius" is returned, and you can replace the `validEntry` method call by the value “ksebelius”. Since the method call is embedded inside an assignment statement, the method call’s returned value, “ksebelius”, is then assigned to the `username` variable.

S6.6 Method Calls Without Data Transfer

void Methods

For a long time, you’ve used this heading for your `main` methods:

```
public static void main(String args[])
```

Note that `main`’s return type is `void`. That means `main` returns nothing, and we say that it’s a “`void` method.” In this section, you’ll learn about `void` method details by examining the MemorizePasswordEntry program. The program is similar to the

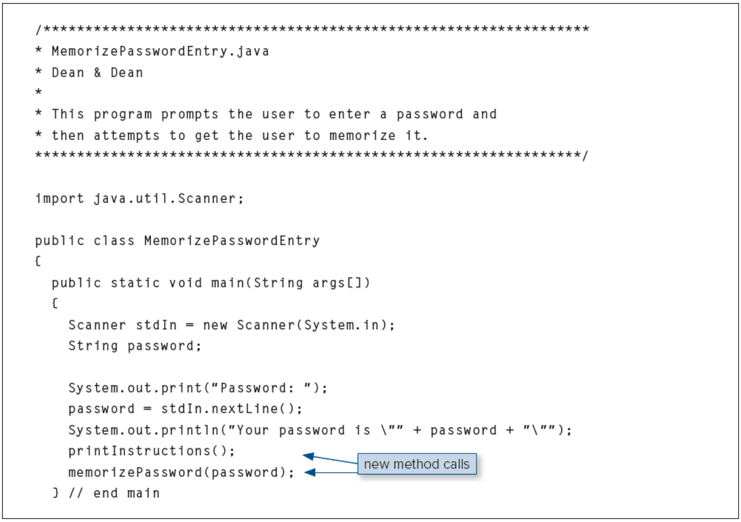
UsernamePasswordEntry program in that it handles user input for a password, but it doesn't bother with a username or input validation. The MemorizePasswordEntry program contains two new methods, whose headings are:

```
  public static void
printInstructions()
  public static void
memorizePassword(String
password)
```

To exercise these two new methods, we insert the following two statements at the end of Figure S6.2a's `main` method:

```
  printInstructions();
  memorizePassword(password);
```

Since these methods are `void` methods and return nothing, their method calls appear on lines by themselves. Remember how the `validEntry` method calls were embedded inside assignment statements? That wouldn't work with `printInstructions` and `memorizePassword` because they return nothing and you can't assign nothing in an assignment statement.



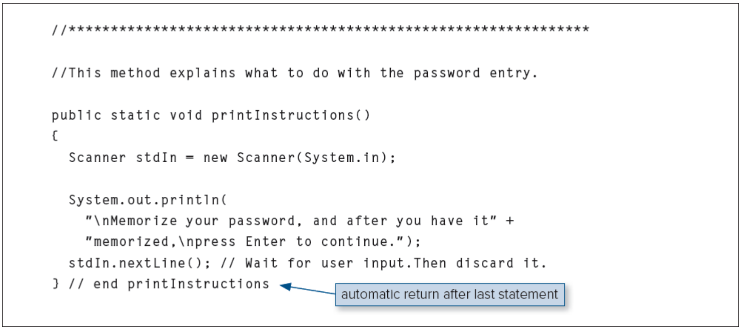
```
/*********************************************************************
* MemorizePasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter a password and
* then attempts to get the user to memorize it.
*********************************************************************/

import java.util.Scanner;

public class MemorizePasswordEntry
{
  public static void main(String args[])
  {
    Scanner stdIn = new Scanner(System.in);
    String password;

    System.out.print("Password: ");
    password = stdIn.nextLine();
    System.out.println("Your password is \"" + password + "\"");
    printInstructions();
    memorizePassword(password);
  } // end main
```

Figure S6.2a MemorizePasswordEntry program—part A

The `printInstructions` method appears in Figure S6.2b. Since its method heading parentheses are empty, the `printInstructions` method defines no parameters, and a method call passes no data in. The `printInstructions` method simply prints this message:

```
  Memorize your password, and
after you have it memorized,
  press Enter to continue.
```

Since `printInstructions` is a `void` method, no data passes out. Note that there is no `return` statement. When there is no `return` statement, a method automatically returns after execution of its last statement.



```
//*****************************************************************

//This method explains what to do with the password entry.

public static void printInstructions()
{
  Scanner stdIn = new Scanner(System.in);

  System.out.println(
    "\nMemorize your password, and after you have it" +
    "memorized,\npress Enter to continue.");
  stdIn.nextLine(); // Wait for user input.Then discard it.
} // end printInstructions
```

automatic return after last statement

Figure S6.2b MemorizePasswordEntry program—part B

Empty return Statement

For methods with a `void` `return` type, it's legal to have an *empty* `return` statement. The empty `return` statement looks like this:

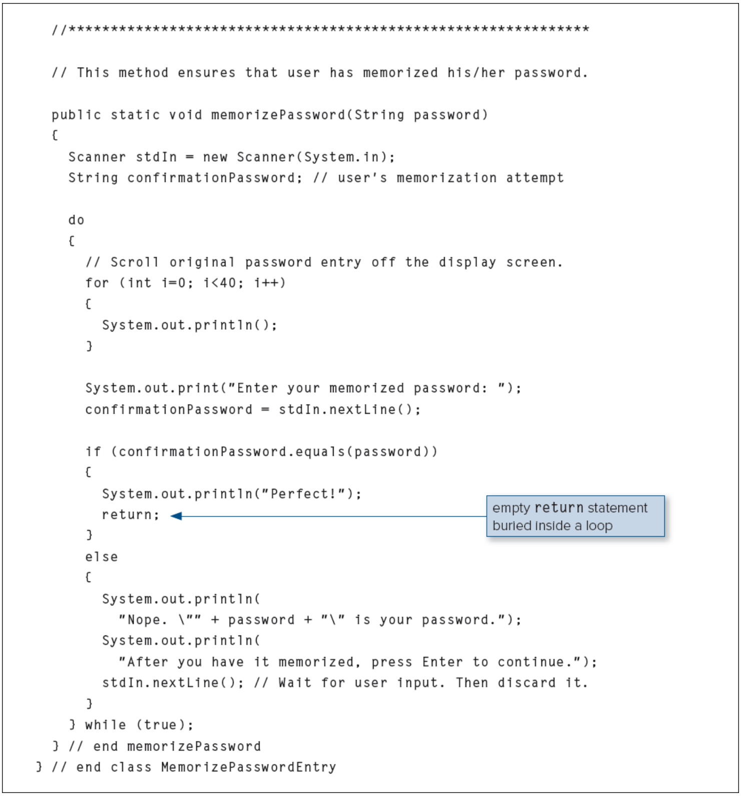
```
return;
```

The empty `return` statement does what you'd expect. It terminates the current method and causes control to be passed back to the calling module at the point that immediately follows the method call.

Note the empty `return` statement inside of Figure S6.2c's `memorizePassword` method. The purpose of `memorizePassword` is to ensure that the user has memorized his/her password. This is done by receiving the password as a `password` parameter, making the previously entered password scroll off the display screen by printing 40 blank lines in a loop, and then prompting the user to enter the memorized password. If the user's entry equals the `password` parameter's value, the program prints "Perfect!" and executes the empty `return` statement. If the user's entry does not equal the `password` parameter's value, the program prints the original password, asks the user to try again to memorize it, and repeats the loop.

The `do` loop's closing condition, `while (true)`, ensures that the loop continues. So what keeps the program from running on forever? What stops the looping is the `return` statement that's buried inside the loop. When the `return` statement executes, control bypasses the closing `while (true);` and returns to MemorizePasswordEntry's `main` method.

When a method includes conditional branching (as in an `if` statement or `switch` statement), it's possible to return from more than one place in the method. In such cases, all returns must match the type specified in the method heading. It would be illegal to have an empty `return` statement and a non-empty `return` statement in the same method. Why? Empty and non-empty `return` statements have different return types (`void` for an empty `return` statement and some other type for a non-empty `return` statement). There is no way to specify a type in the heading that simultaneously matches two different return types.



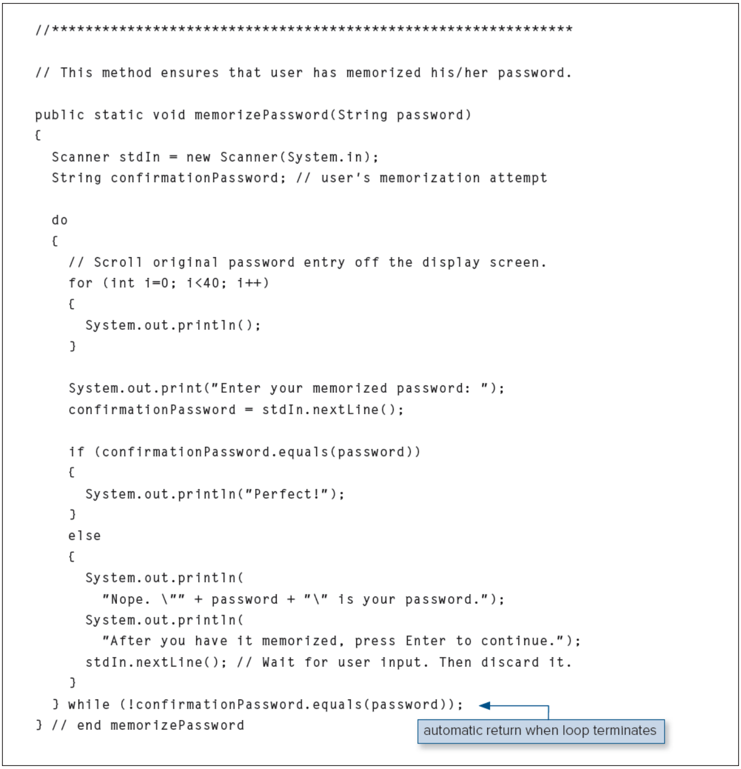
```
    //*****************************************************************

    // This method ensures that user has memorized his/her password.

    public static void memorizePassword(String password)
    {
      Scanner stdIn = new Scanner(System.in);
      String confirmationPassword; // user's memorization attempt

      do
      {
        // Scroll original password entry off the display screen.
        for (int i=0; i<40; i++)
        {
          System.out.println();
        }

        System.out.print("Enter your memorized password: ");
        confirmationPassword = stdIn.nextLine();

        if (confirmationPassword.equals(password))
        {
          System.out.println("Perfect!");
          return;
        }
        else
        {
          System.out.println(
            "Nope. \"" + password + "\" is your password.");
          System.out.println(
            "After you have it memorized, press Enter to continue.");
          stdIn.nextLine(); // Wait for user input. Then discard it.
        }
      } while (true);
    } // end memorizePassword
  } // end class MemorizePasswordEntry
```

empty `return` statement buried inside a loop

Figure S6.2c MemorizePasswordEntry program—part C

The empty `return` statement is a helpful statement in that it provides an easy way to exit quickly from a `void` method. However, it does not provide unique functionality. Code that uses an empty `return` statement can always be replaced by code that has no `return` statements. For example, Figure S6.3’s `memorizePassword` method is functionally equivalent to Figure S6.2c’s `memorizePassword` method, but Figure S6.3’s version has no `return` statement. It implements the return the old-fashioned way—by allowing the loop to terminate via the loop’s termination condition and allowing the method to finish executing. When the method finishes executing, control will automatically return to where the method was called. Notice how the loop terminates by replacing the original `do` loop’s closing condition, `while (true)`, with this:

```
} while (!confirmationPassword.equals(passw
```



```
//******************************************************************

// This method ensures that user has memorized his/her password.

public static void memorizePassword(String password)
{
  Scanner stdIn = new Scanner(System.in);
  String confirmationPassword; // user's memorization attempt

  do
  {
    // Scroll original password entry off the display screen.
    for (int i=0; i<40; i++)
    {
      System.out.println();
    }

    System.out.print("Enter your memorized password: ");
    confirmationPassword = stdIn.nextLine();

    if (confirmationPassword.equals(password))
    {
      System.out.println("Perfect!");
    }
    else
    {
      System.out.println(
        "Nope. \"" + password + "\" is your password.");
      System.out.println(
        "After you have it memorized, press Enter to continue.");
      stdIn.nextLine(); // Wait for user input. Then discard it.
    }
  } while (!confirmationPassword.equals(password));
} // end memorizePassword
```

Figure S6.3 Alternate way to return from memorizePassword method

return Statement Within a Loop

Programmers in industry often are asked to maintain (fix and improve) other people’s code. In doing that, they often find themselves having to examine the loops and, more specifically, the loop termination conditions in the program they're working on. Therefore, it's important that loop termination conditions be clear. Normally, loop termination conditions appear in the standard loop-condition location. For `while` loops, that's at the top, for `do` loops that's at the bottom, and for `for` loops, that's in the header's second component. However, a `return` statement inside a loop results in a loop termination condition that's not in a standard location. For example, in our first `memorizePassword` method (in Figure S6.2c), the `return` statement is inside an `if` statement and the loop termination condition is consequently "hidden" in the `if` statement's condition.

In the interest of maintainability, you should use restraint when considering the use of a `return` statement inside a loop. Based on the context, if inserting `return` statements inside a loop improves clarity, then feel free to insert. However, if it simply makes the coding chores easier and it does not add clarity, then don't insert. So, which `memorizePassword` implementation is better—the empty `return` version or the `return`-less version? In general, we prefer the `return`-less version for maintainability reasons. However, because the code in both of our `memorizePassword` methods is relatively simple, it doesn't make much of a difference here.

S6.7 Argument Passing

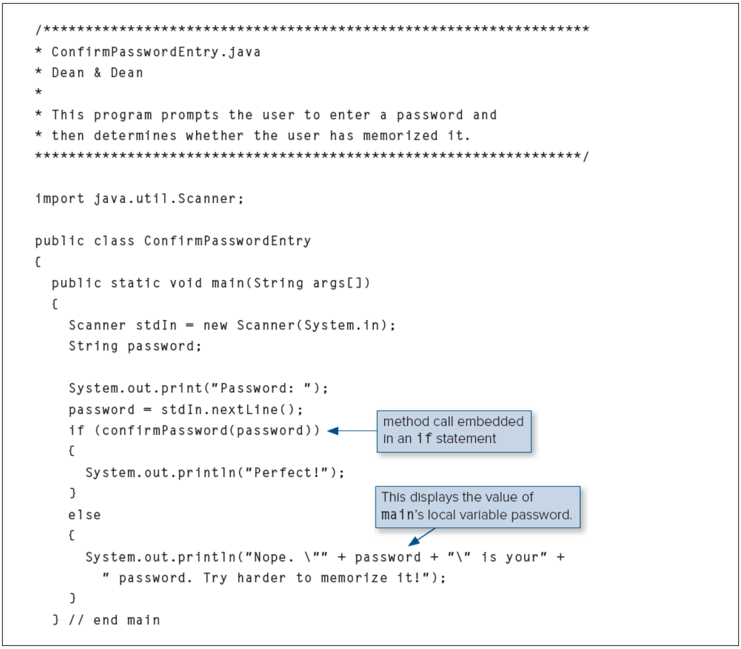
You've worked with argument passing for quite a while now, but there are a few more details that you should be aware of. In this section, we discuss the pass-by-value scheme for passing arguments, and we also discuss using the same name versus using different names for argument-parameter pairs.

Example

Let's dig into the details of argument passing by examining another program, ConfirmPasswordEntry. The purpose of ConfirmPasswordEntry is to determine whether the user has memorized his/her entered password. There's no loop this time (as there is in the MemorizePasswordEntry program), so if the user guesses wrong, that's it, there are no second guesses.

See Figure S6.4a, which shows the program's `main` method. Within the `main` method, note the call to `confirmPassword`. That method call is embedded in the condition part of an `if` statement. That should make sense when you realize that the `confirmPassword` method has a return type of `boolean`. As shown in Figure S6.4b, the `confirmPassword` method returns `true` if the user guesses the password correctly and returns `false` otherwise. The returned `true` or `false` value is then used as the condition in main's `if` statement.

ConfirmPasswordEntry's `main` method calls the `confirmPassword` method with an argument named `password`. The `password` argument holds the user's original password entry. As part of the method call process, `main`'s `password` argument value gets assigned to the `confirmPassword` method's `password` parameter, as shown in Figure S6.4b. The `confirmPassword` method then makes the previously entered password scroll off the display screen by printing 40 blank lines in a loop. The `confirmPassword` method then prompts the user to guess the memorized password. Here's the key—the user's guess gets assigned into the `password` parameter. What happens to the corresponding `password` argument variable in the `main` method if the `password` parameter's value changes? Will there be a simultaneous change in `main`'s `password` argument variable? Because the `password` argument and the `password` parameter are distinct variables, the `password` variable in `main` does not change with the `password` variable in `confirmPassword`. So as shown in Figure S6.4b's sample session, when `main` prints its version of `password` at the end of the program, it prints the unchanged original password value, “uwmpanther4”, not the “panther4” value stored in `confirmPassword`'s `password` variable.



```
/*******************************************************************
* ConfirmPasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter a password and
* then determines whether the user has memorized it.
*******************************************************************/

import java.util.Scanner;

public class ConfirmPasswordEntry
{
  public static void main(String args[])
  {
    Scanner stdIn = new Scanner(System.in);
    String password;

    System.out.print("Password: ");
    password = stdIn.nextLine();
    if (confirmPassword(password))
    {
      System.out.println("Perfect!");
    }
    else
    {
      System.out.println("Nope. \"" + password + "\" is your" +
        " password. Try harder to memorize it!");
    }
  } // end main
```

method call embedded in an `if` statement

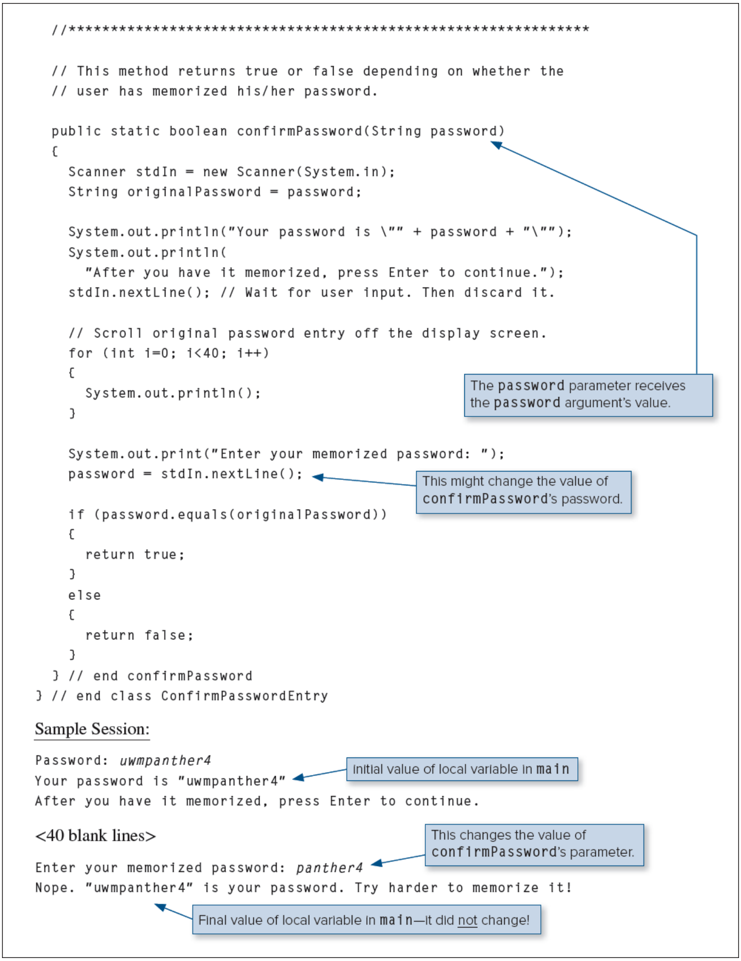
This displays the value of `main`'s local variable password.

Figure S6.4a ConfirmPasswordEntry program—part A

Pass-by-Value

We say that Java uses *pass-by-value* for its argument-passing scheme. As illustrated by Figure S6.5, pass-by-value means that the JVM passes a copy of the argument's value (not the argument itself) to the parameter. Changing the copy does not change the original.

Notice that the calling method's argument is called `password` and the `confirmPassword` method's parameter is called `password` also. Does using the same name make them the same thing? No! They are separate variables separately encapsulated in separate blocks of code. Because these two variables are in separate blocks of code, there is no conflict, and it's OK to give them the same name. Using the same name is natural because these two variables describe the same kind of thing. When names are in different blocks, you don't have to worry about whether they are the same or not. That's the beauty of encapsulation. Big programs would be horrible nightmares if you were prohibited from using the same name in different blocks of code.



```
//***************************************************************

// This method returns true or false depending on whether the
// user has memorized his/her password.

public static boolean confirmPassword(String password)
{
  Scanner stdIn = new Scanner(System.in);
  String originalPassword = password;

  System.out.println("Your password is \"" + password + "\"");
  System.out.println(
    "After you have it memorized, press Enter to continue.");
  stdIn.nextLine(); // Wait for user input. Then discard it.

  // Scroll original password entry off the display screen.
  for (int i=0; i<40; i++)
  {
    System.out.println();
  }

  System.out.print("Enter your memorized password: ");
  password = stdIn.nextLine();

  if (password.equals(originalPassword))
  {
    return true;
  }
  else
  {
    return false;
  }
} // end confirmPassword
} // end class ConfirmPasswordEntry
```

The `password` parameter receives the `password` argument's value.

This might change the value of `confirmPassword`'s password.

Sample Session:

```
Password: uwmpanther4
Your password is "uwmpanther4"
After you have it memorized, press Enter to continue.
```

Initial value of local variable in `main`

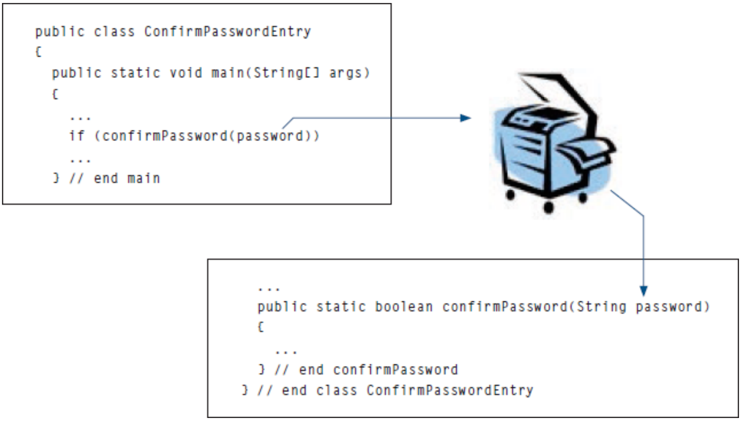
<40 blank lines>

```
Enter your memorized password: panther4
Nope. "uwmpanther4" is your password. Try harder to memorize it!
```

This changes the value of `confirmPassword`'s parameter.

Final value of local variable in `main`—it did not change!

Figure S6.4b ConfirmPasswordEntry program—part B



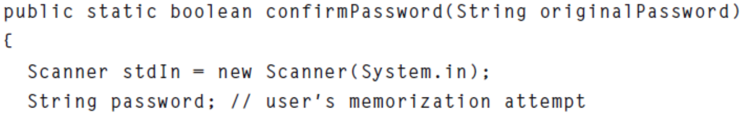
```
public class ConfirmPasswordEntry
{
  public static void main(String[] args)
  {
    ...
    if (confirmPassword(password))
    ...
  } // end main
```

```
  ...
  public static boolean confirmPassword(String password)
  {
    ...
  } // end confirmPassword
} // end class ConfirmPasswordEntry
```

Figure S6.5 Pass-by-value means a copy of the argument’s value goes to the corresponding

Same Name Versus Different Names for Argument-Parameter Pairs

So far in this chapter, all of our programs have used the same name for argument-parameter pairs because we want to encourage you to take advantage of encapsulation and use the most natural and reasonable names whenever you can. But we don’t want you to think that argument and parameter names must be the same always. When it's more natural and reasonable to use different names for an argument-parameter pair, then use different names. The only requirement is that the argument's type must match the parameter's type. For example, suppose you felt that "originalPassword" would be a more descriptive name for the parameter in `confirmPassword`'s heading. To implement that feeling and implement a functionally equivalent `confirmPassword` method, all you'd have to do is replace the method's first four lines with these four lines:



```
public static boolean confirmPassword(String originalPassword)
{
  Scanner stdIn = new Scanner(System.in);
  String password; // user's memorization attempt
```

The compiler would have no problem with the `originalPassword` parameter in `confirmPassword` and the `password` argument in `main` having different names because the parameter and argument would be the same type—`String`.

In the above code fragment, `password` is a local variable. As such, it would not interfere with `main`’s `password` variable. When the user’s password guess gets assigned into `confirmPassword`’s `password` variable (as shown in Figure S6.4b), `main`’s `password` variable would not be affected. Yeah!

S6.8 Tracing a Program with More than One Method

To reinforce what you’ve learned so far, we’ll trace a simplified version of our UsernamePasswordEntry program. Remember the tracing procedure we used in prior chapters? It worked fine for programs with only one method—the `main` method. But for multiple methods, you’ll need to keep track of which method you’re in. In addition, you'll need to keep track of parameters and local variables in subordinate methods. This requires a more elaborate trace table.

The first step in performing a trace is to generate a copy of the code that includes line numbers. These line numbers provide the necessary connection between individual trace events and the code statements that caused those events. Figure S6.6 contains the UserPasswordEntryTrace program with line numbers.

Trace Setup

Figure 6.7 shows the setup. As with previous traces, the input goes in the top-left corner. To illustrate the validation loop's functionality, we have inserted an invalid password entry before the final valid password entry.

Unlike traces in previous chapters, the trace-table headings now require more than one line. The first heading line has class names. Because both methods are `public`, they could be in different classes (with separate line numbers for each class), but we elected to put them both in the same `UserNamePasswordEntryTrace` class. Under each class name heading, there's a heading for each of that class's methods. Under each method-name heading, there's a heading for each of that method's parameters and local variables. The two entries under `main` are `main`'s two local variables (`username` and `password`). The first two entries under `validEntry` are its parameters (`minLength` and `entryType`), and the third entry under `validEntry` is its local variable (`entry`). The wide column on the right is for output.

The `validEntry` method also defines another local variable, `stdIn`, but we don't need to trace it because it uses Java API components. They've already been traced and tested thoroughly by the implementers of the Java programming language, and you can assume they work properly.

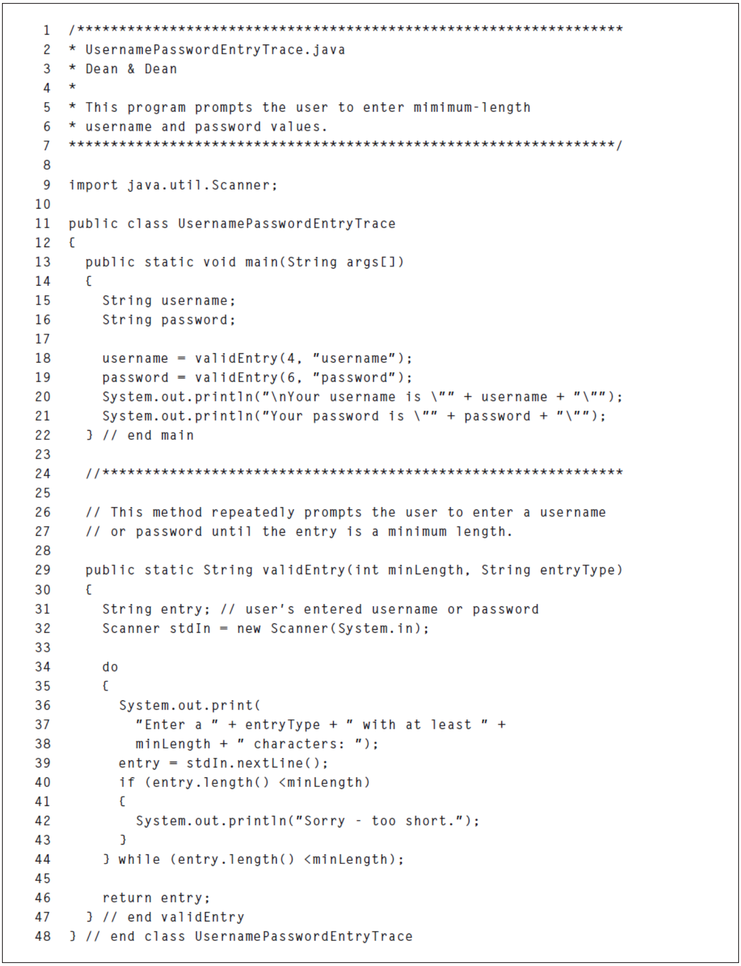
Trace Execution

Using Figure S6.7's trace setup as a starting point, we'll walk you through the key sections of the trace shown in Figure S6.8. When declaring a local variable (as on lines 15 and 16 in Figure S6.6), enter the code line number and initial values in appropriate columns, using "?" if not initialized. When calling a method (as on lines 18 and 19), enter the code line number at which the call occurs, and under the method's parameter headings, enter the passed in parameter values. When printing output (as on lines 20 and 21), enter the code line number and the output's literal and variable values in the output column.

When making an assignment, enter the code line number, and for the variable to which the assignment is made, enter the assigned value. Similarly, when reading input (as on line 39) enter the code line number, and for the variable getting the input, copy the value from the next item in the input list at the upper left hand corner of the trace table. (To keep track of where you are in the input list, you might want to put a check mark after each input item as you use it.)

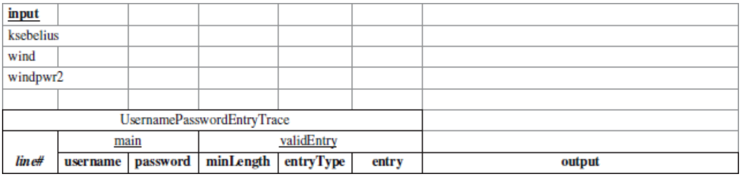
Except at the start of a `for` loop (where you need to initialize and update the `for` loop index variable), don't make entries at branch points. Just look at the data already in the trace table to decide where to go next. For example, at the `if` condition on code line 40, ask yourself, "Is the length of the string in the `entry` variable less than the value stored in `minLength`?" If yes, go to code line

42. If no, go to code line 44.



```
 1  /*******************************************************************
 2  * UsernamePasswordEntryTrace.java
 3  * Dean & Dean
 4  *
 5  * This program prompts the user to enter mimimum-length
 6  * username and password values.
 7  *******************************************************************/
 8
 9  import java.util.Scanner;
10
11  public class UsernamePasswordEntryTrace
12  {
13    public static void main(String args[])
14    {
15      String username;
16      String password;
17
18      username = validEntry(4, "username");
19      password = validEntry(6, "password");
20      System.out.println("\nYour username is \"" + username + "\"");
21      System.out.println("Your password is \"" + password + "\"");
22    } // end main
23
24    //*****************************************************************
25
26    // This method repeatedly prompts the user to enter a username
27    // or password until the entry is a minimum length.
28
29    public static String validEntry(int minLength, String entryType)
30    {
31      String entry; // user's entered username or password
32      Scanner stdIn = new Scanner(System.in);
33
34      do
35      {
36        System.out.print(
37          "Enter a " + entryType + " with at least " +
38          minLength + " characters: ");
39        entry = stdIn.nextLine();
40        if (entry.length() <minLength)
41        {
42          System.out.println("Sorry - too short.");
43        }
44      } while (entry.length() <minLength);
45
46      return entry;
47    } // end validEntry
48  } // end class UsernamePasswordEntryTrace
```

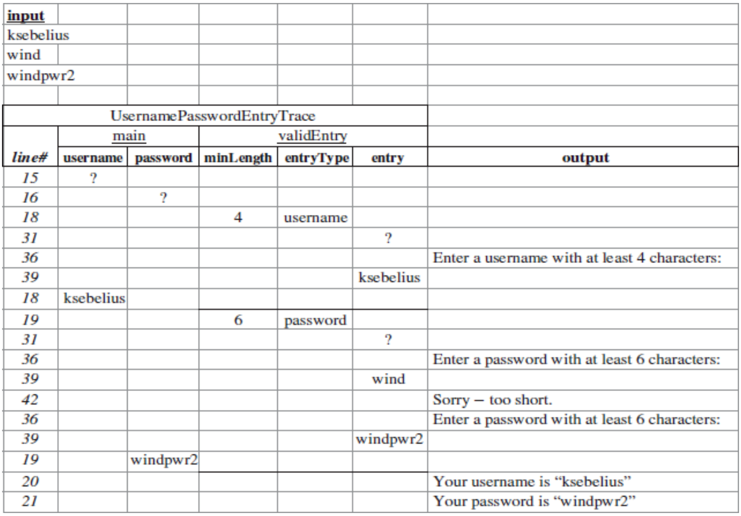
Figure S6.6 Numbered UsernamePasswordEntryTrace program



input						
ksebelius						
wind						
windpwr2						
	UsernamePasswordEntryTrace					
	main		validEntry			
line#	username	password	minLength	entryType	entry	output

Figure S6.7 Trace setup for the

UsernamePasswordEntryTrace program



input						
ksebelius						
wind						
windpwr2						
UsernamePasswordEntryTrace						
	main		validEntry			
line#	**username**	**password**	**minLength**	**entryType**	**entry**	**output**
15	?					
16		?				
18			4	username		
31					?	
36						Enter a username with at least 4 characters:
39					ksebelius	
18	ksebelius					
19			6	password		
31					?	
36						Enter a password with at least 6 characters:
39					wind	
42						Sorry – too short.
36						Enter a password with at least 6 characters:
39					windpwr2	
19		windpwr2				
20						Your username is “ksebelius”
21						Your password is “windpwr2”

Figure S6.8 Completed trace of UsernamePasswordEntry program

To indicate a `return` from a method, go to a new row on the trace table and draw a heavy horizontal line under all the parameters and local variables in that method. This heavy underline says these variables are wiped out after the `return`. On that same row in the trace table, enter the code line number of the point in the calling method to which control returns.2 In our UserPasswordEntryTrace program, that's code line 18 after the first `validEntry` call and code line 19 after the second `validEntry` call. If there is a return value (i.e., the called method is not a `void` method) on this same row in the trace table, enter the return value under the variable to which the return value is assigned. In our example, after the first `validEntry` call, on code line 18 the returned `ksebelius` value is assigned to `username`. After the second `validEntry` call, on code line 19 the returned `windpwr2` value is assigned to `password`.

Experience with this long-form tracing technique will make it easier for you to understand what an automated debugger in an *Integrated Development Environment* (IDE) is telling you. As you step through a program that's running in debug mode under the control of an IDE debugger, when you get to a method call, you have two choices. You can "step into" and go through all the statements in the called method, like we do in Figure S6.8, or you can "step over" and just see what happens after the method returns. If our trace table had stepped over both `validEntry` method calls, it would have had a total of only eight rows, for code lines 15, 16, 18, 18, 19, 19, 20, and 21. In a typical debugging activity, you will use a combination of stepping over and stepping into.

S6.9 Overloaded Methods

Up until this point, all of the methods we defined for a given class have had unique names. But if you think back to some of the Java API methods presented in Chapter 5, you'll recall that there were several examples where the same name (`abs`, `max`, `min`) was used to identify more than one method in the same class (the

`Math` class). This section will show you how to do this in classes you write.

What Are Overloaded Methods?

Overloaded methods are two or more methods in the same class that use the same name. Since they use the same name, the compiler needs something else besides the name to distinguish them. Parameters to the rescue! To make two overloaded methods distinguishable, you define them with different parameters. More specifically, you define them with a different number of parameters or different types of parameters. The combination of a method's name, the number of its parameters, and the types of its parameters is called the method's *signature*. Each distinct method has a distinct signature. Could these three lines be used as headings for three overloaded `findMaximum` methods?

```
  int findMaximum(int a, int b, int c)
  double findMaximum(double a, double b, double c)
  double findMaximum(double a, double b, double c, double d)
```

Yes, they are a legal overloading of the `findMaximum` method name, because each heading is distinguishable in terms of number and types of parameters. How about the next two lines—could a `findAverage` method name be overloaded in this way?

```
  int findAverage(int a, int b, int c)
  double findAverage(int x, int y, int z)
```

No. These are not distinguishable methods, because they have the same signature—same method names and same number and types of parameters. Since these two methods are not distinguishable, if you try to include these two method headings in the same class, the compiler will think you're defining the same method twice. And that will make the compiler irritable. Be prepared for it to snarl back at you with a "duplicate definition" compile-time error message.

Note that the above `findAverage` method headings have different return types. You might think that the different return types indicate different signatures. Not true. The return type is not part of the signature, so you cannot use just a different return type to distinguish overloaded methods.

Benefit of Overloaded Methods

When should you use overloaded methods? When there's a need to perform essentially the same task with different parameters. For example, the methods associated with the above `findMaximum` headings perform essentially the same basic task—they calculate the maximum value from a given list of numbers. But they perform the task on different sets of parameters. Given that situation, overloaded methods are a perfect fit.

Note that using overloaded methods is never an absolute requirement. As an alternative, you can always use different method names to distinguish different methods. So why are the above `findMaximum` method headings better than the below method headings?

```
  int findMaximumOf3Ints(int a, int b, int c)
  double findMaximumOf3Doubles(double a, double b, double c)
  double findMaximumOf4Doubles(double a, double b, double c, double d)
```

As these examples suggest, using different method names is cumbersome. With only one method name, the name can be simple. As a programmer, wouldn’t you prefer to use and remember just one simple name rather than several cumbersome names?

Example

Suppose you want methods that compute the distance between two points in space, where the space might have either one dimension (be along a line) or two dimensions (be on a plane). In the future, you might want to add similar methods to determine distance between two points in three dimensions (in a volume) or perhaps even more than three dimensions (in “hyperspace”). In principal, you can get what you need immediately and also be ready for the future by settling on one name for all possibilities, `findDistance`. Then let the dimensionality establish the number of parameters.

The one dimensional method could have a heading like this:

```
  public static double
findDistance(double x1,
double x2)
```

In this one-dimensional method, `x1` gives the position of the first point, and `x2` gives the position of the second point.

The two dimensional method could have a heading like this:

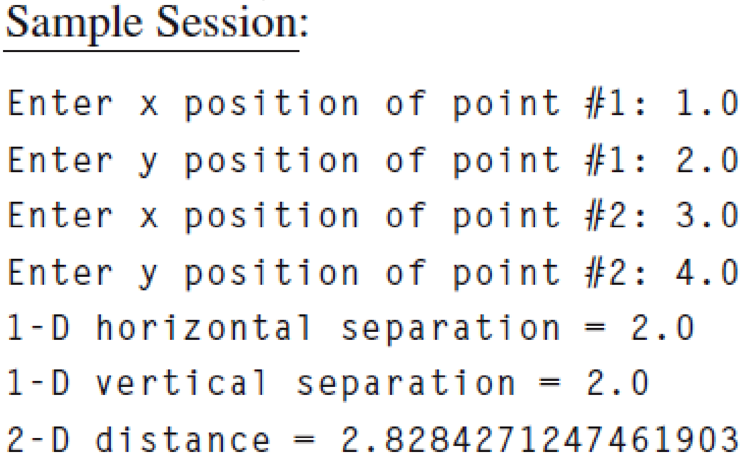
```
  public static double
findDistance(
  double x1, double y1,
double x2, double y2)
```

In this two-dimensional method, `x1` and `y1` give the position of the first point, and `x2` and `y2` give the position of the second point.

These two `findDistance` methods

perform essentially the same task—they determine the distance between two points. And they do it with only a slight variation—the dimensionality of the space in which the points exist. That's why we use the same name and "overload" that name.

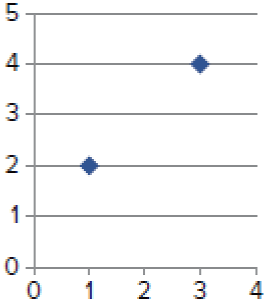
Here's an example of what a program running these overloaded methods might do:



Sample Session:

```
Enter x position of point #1: 1.0
Enter y position of point #1: 2.0
Enter x position of point #2: 3.0
Enter y position of point #2: 4.0
1-D horizontal separation = 2.0
1-D vertical separation = 2.0
2-D distance = 2.8284271247461903
```

Here's a picture showing this example's location of the two points in the x-y plane (x is horizontal and y is vertical):



With equal horizontal and vertical separations, the straight-line distance should equal the separation in either direction times √–2 = 1.414…; and since 2.0 × 1.414… = 2.828…, it does.

Figures S6.9a and S6.9b contain a program that defines and uses the desired overloaded methods to generate the above results. Figure S6.9a's `main` method makes two calls to the two-parameter version of `findDistance` to compute the separations in the horizontal (x) and vertical(y) directions. Then it calls the four-parameter version to determine the direct distance between the two points. Figure S6.9b presents straightforward implementations of the overloaded `findDistance` methods.

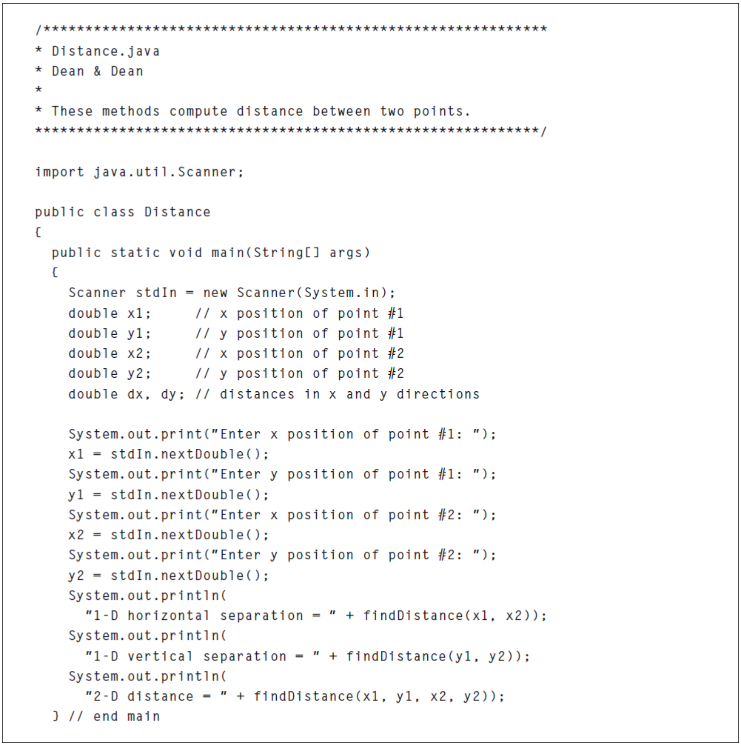
Calling an Overloaded Method from Within Another Overloaded Method

Look at the two implementations of the overloaded `findDistance` methods in Figure S6.9b. In the two-dimensional (4-parameter) version, the local variables, `dx` and `dy`, are initialized with a mathematical operation that is similar to the mathematical operation used to implement the one-dimensional (2-parameter) version of the method. This suggests calling the 2-parameter method inside the 4-parameter method, like this:

```
public static double findDistance(
  double x1, double y1, double x2, double y2)
{
  double dx = findDistance(x1, x2);
  double dy = findDistance(y1, y2);
  return Math.sqrt(dx * dx + dy * dy);
} // end findDistance in 2-D
```

This variation of the 4-parameter

`findDistance` method produces exactly the same results as the 4-parameter `findDistance` method in Figure S6.9b. The latter version shows that an overloaded method can call another overloaded method. Additional method calls make the latter implementation slightly less efficient, but one might consider it more elegant because it eliminates code redundancy.



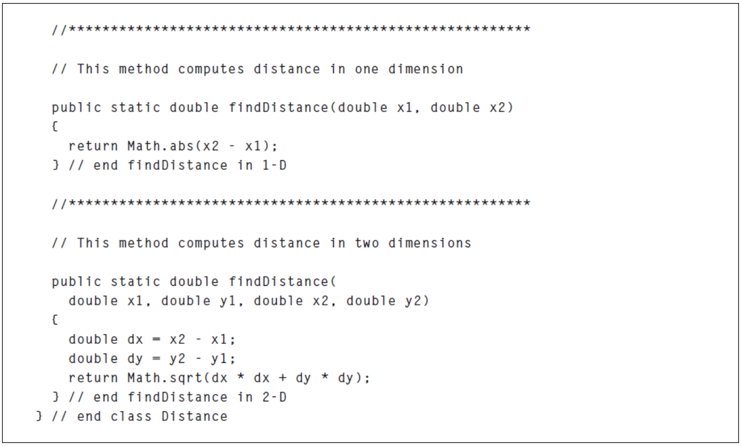
```
/*************************************************************
* Distance.java
* Dean & Dean
*
* These methods compute distance between two points.
*************************************************************/

import java.util.Scanner;

public class Distance
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    double x1;     // x position of point #1
    double y1;     // y position of point #1
    double x2;     // x position of point #2
    double y2;     // y position of point #2
    double dx, dy; // distances in x and y directions

    System.out.print("Enter x position of point #1: ");
    x1 = stdIn.nextDouble();
    System.out.print("Enter y position of point #1: ");
    y1 = stdIn.nextDouble();
    System.out.print("Enter x position of point #2: ");
    x2 = stdIn.nextDouble();
    System.out.print("Enter y position of point #2: ");
    y2 = stdIn.nextDouble();
    System.out.println(
      "1-D horizontal separation = " + findDistance(x1, x2));
    System.out.println(
      "1-D vertical separation = " + findDistance(y1, y2));
    System.out.println(
      "2-D distance = " + findDistance(x1, y1, x2, y2));
  } // end main
```

Figure S6.9a Distance program—part A:

main method

Program Evolution

The ability to overload a method name promotes graceful program evolution, because it corresponds to how natural language regularly overloads the meanings of words. For example, the first version of a program might define a one-parameter version of some particular method. Later, when you decide to enhance your program, it's easier for your existing users if you minimize the new things they have to learn. You let them either keep using the original method or switch to the improved method, which has an additional parameter that enables selection among several options. When they want to use the improved method, all they have to remember is the original method name with another parameter that provides options. That's an almost obvious variation, and it's easier to remember than a different method name. It’s certainly easier than being forced to learn a new method name for the old task—which would be a necessary cost of upgrading if method overloading were not available.



```
  //***********************************************************

  // This method computes distance in one dimension

  public static double findDistance(double x1, double x2)
  {
    return Math.abs(x2 - x1);
  } // end findDistance in 1-D

  //***********************************************************

  // This method computes distance in two dimensions

  public static double findDistance(
    double x1, double y1, double x2, double y2)
  {
    double dx = x2 - x1;
    double dy = y2 - y1;
    return Math.sqrt(dx * dx + dy * dy);
  } // end findDistance in 2-D
} // end class Distance
```

Figure S6.9b Distance program—part B: overloaded `findDistance` methods

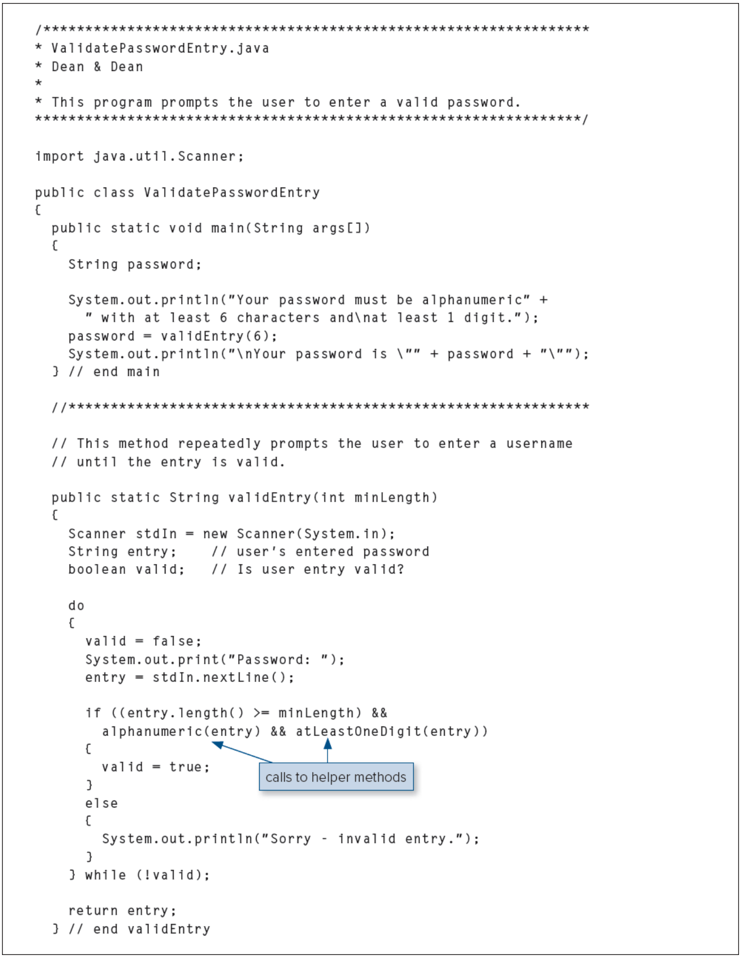
S6.10 Helper Methods

As problems get bigger, it becomes more and more necessary to partition them into sub-problems, each of which has a manageable size. We started doing this in Chapter 5 when our `main` method called on some of Java's API methods for help. In this chapter, we've been writing programs with multiple methods, where the `main` method calls one or more subordinate methods. In a broad sense, any method that is called by another method is a "helper method" because the called method helps the calling method. But most programmers prefer a narrower definition for a helper method. We'll now explain the narrower definition….

Up to this point, all methods you've used have been `public`; that is, they've been defined with the `public` access modifier. These `public` methods are part of the class's *interface*, because they handle the communication between the class and the outside world. Sometimes, you'll want to create a method that is not part of the interface. Instead it just supports the operation of other non-

`main` methods within its own class. Those types of methods are commonly referred to as *helper methods*, and they can be recognized by their use of the `private` access modifier.

To further your understanding of helper methods, let's use them to write an improved version of the `validEntry` method. Glance back at Figure S6.1b, and you'll see the somewhat cumbersome code that comprises the original `validEntry` method. Now take a look at the new and improved `validEntry` method in Figure S6.10a.

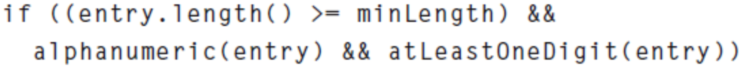


```
/*******************************************************************
* ValidatePasswordEntry.java
* Dean & Dean
*
* This program prompts the user to enter a valid password.
*******************************************************************/

import java.util.Scanner;

public class ValidatePasswordEntry
{
  public static void main(String args[])
  {
    String password;

    System.out.println("Your password must be alphanumeric" +
      " with at least 6 characters and\nat least 1 digit.");
    password = validEntry(6);
    System.out.println("\nYour password is \"" + password + "\"");
  } // end main

  //****************************************************************

  // This method repeatedly prompts the user to enter a username
  // until the entry is valid.

  public static String validEntry(int minLength)
  {
    Scanner stdIn = new Scanner(System.in);
    String entry;    // user's entered password
    boolean valid;   // Is user entry valid?

    do
    {
      valid = false;
      System.out.print("Password: ");
      entry = stdIn.nextLine();

      if ((entry.length() >= minLength) &&
        alphanumeric(entry) && atLeastOneDigit(entry))
      {
        valid = true;
      }
      else
      {
        System.out.println("Sorry - invalid entry.");
      }
    } while (!valid);

    return entry;
  } // end validEntry
```

calls to helper methods

Figure S6.10a ValidatePasswordEntry program—part A

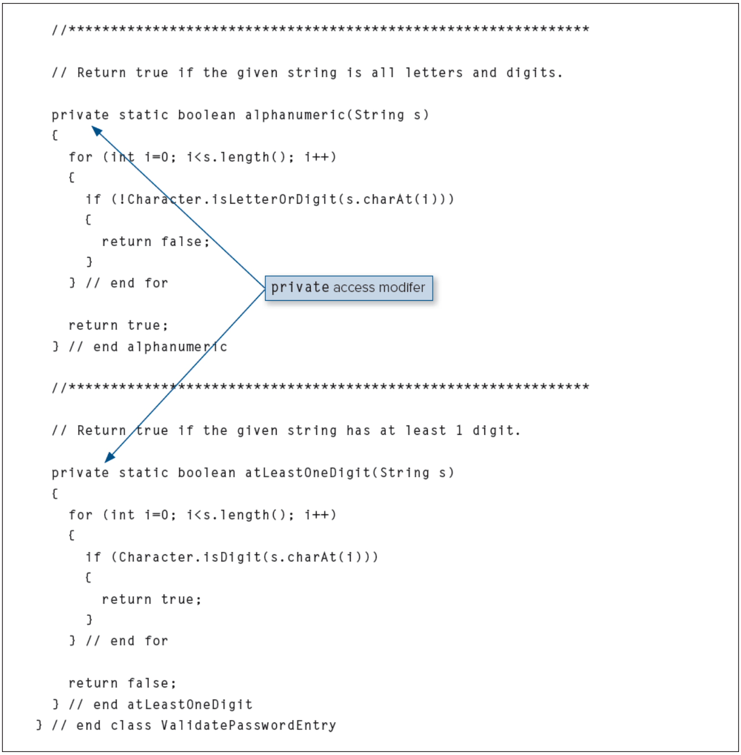
In the interest of keeping things simple, the new `validEntry` method handles just passwords (and not usernames), so there's no need for the original method's `entryType` parameter. But other than that minor difference, the new `validEntry` method is functionally equivalent to the original. The new `validEntry` method is quite a bit shorter than the original, so how can it be functionally equivalent? The key is moving some of `validEntry`’s coding chores to helper methods. The original `validEntry` method included a loop that checked whether the given entry consisted of all alphanumeric characters and whether the given entry contained at least one digit. In the new `validEntry` method, those two tasks are implemented with calls to helper methods. Specifically, note the `alphanumeric(entry)` and `atLeastOneDigit(entry)` method calls in `validEntry`’s `if` statement, copied here for your convenience:



```
if ((entry.length() >= minLength) &&
  alphanumeric(entry) && atLeastOneDigit(entry))
```

That code is fairly easy to understand, and it improves the overall readability of the `validEntry` method.

Now take a look at the helper methods themselves in Figure S6.10b. There's no special syntax for a helper method other than the `private` access modifier in a helper method's heading. The `alphanumeric` method receives a string parameter named `s`, loops through each character in `s`, and returns `false` if and when it finds a character that is not a letter or a digit. Otherwise it returns `true`. The `atLeastOneDigit` method receives a string parameter named `s`, loops through each character in `s`, and returns `true` if and when it finds a character that is a digit. Otherwise it returns `false`. The new `validEntry` method is less efficient than the original `validEntry` method because of the two method calls and the two loops (the original `validEnty` method required just one loop). However, this small, unnoticeable decrease in efficiency is offset by the improvement in readability.



```
  //***************************************************************

  // Return true if the given string is all letters and digits.

  private static boolean alphanumeric(String s)
  {
    for (int i=0; i<s.length(); i++)
    {
      if (!Character.isLetterOrDigit(s.charAt(i)))
      {
        return false;
      }
    } // end for

    return true;
  } // end alphanumeric

  //***************************************************************

  // Return true if the given string has at least 1 digit.

  private static boolean atLeastOneDigit(String s)
  {
    for (int i=0; i<s.length(); i++)
    {
      if (Character.isDigit(s.charAt(i)))
      {
        return true;
      }
    } // end for

    return false;
  } // end atLeastOneDigit
} // end class ValidatePasswordEntry
```

`private` access modifer

Figure S6.10b ValidatePasswordEntry program—part B

---

[1] When something occurs “right off the bat,” it occurs immediately. The idiom originates from baseball. Just after a batter hits a baseball, the baseball will be “right off the bat” and it will tend to

be fast. Such fastness implies immediacy.

[2] If calling and called methods are in separate classes, each class will probably have its own *line* # column. Then if the called method returned from an explicit `return` statement, you would also enter the line number of this `return` statement in the called method's class's *line* # column. This can help you untangle poorly structured returns.

Arrays in a Non-Object-Oriented Environment

[This document is an alternate, non-OOP version of the second half of Chapter 9. It should be read after reading Sections 1–6 in the standard Chapter 9.]

S9.7 Searching an Array

In order to use an array, you need to access its individual elements. If you know the location of the element you're interested in, then you simply access the element by putting the element's index inside square brackets. But if you don't know the location of the element, then you need to search for it. For example, suppose you're writing a program that keeps track of student enrollments for the courses at your school. The program is supposed to be able to add a student, remove a student, view a student's data, and so on. All of those operations require that you first search for the student within a students array (even the add-a-student operation requires a search, to ensure that the student isn't already in the array). In this section, we present two techniques for searching an array.

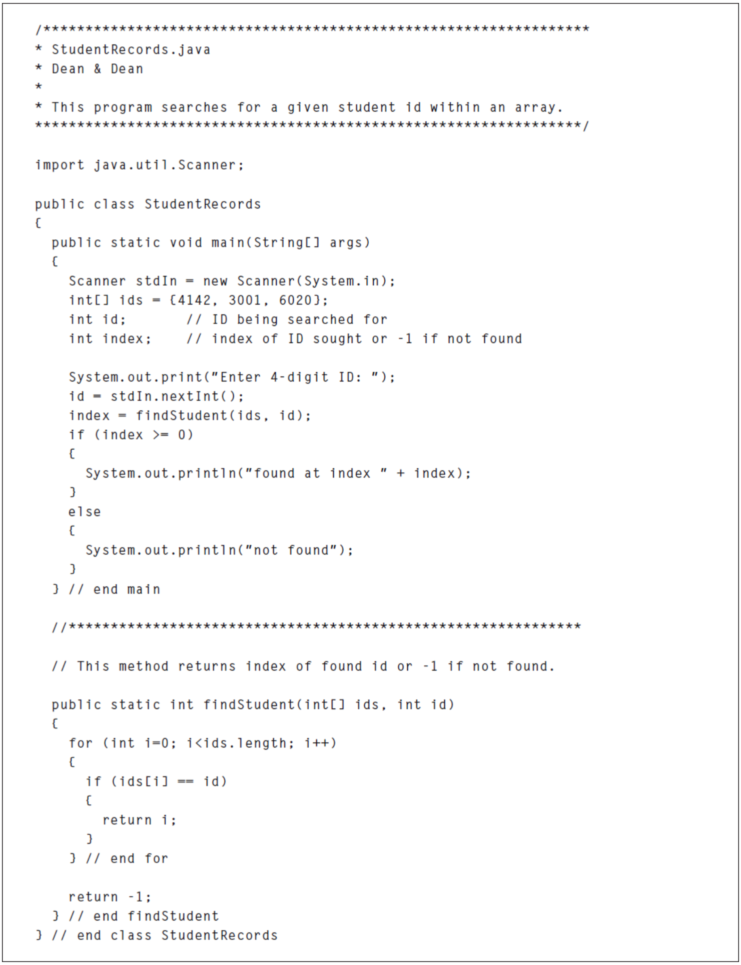
Sequential Search

If the array is short (has less than

about 20 items), the best way to search it is the simplest way: Step through the array sequentially and compare the value at each array element with the searched-for value. When you find a match, do something and return. Here's a pseudocode description of the sequential-search algorithm:

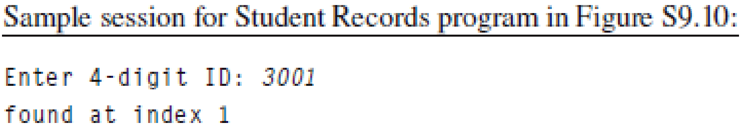
```
i ← 0
while i < number of filled elements
{
   if list[i] equals the searched-for value
      <do something and stop the loop>
   increment i
}
```

Typically, algorithms are more generic than Java implementations. Part of problem solving is the process of adapting generic algorithms to specific situations. In this case, the "do something" code will be different for different cases. Figure S9.10 illustrates an implementation of the sequential-search algorithm. The `main` method initializes a three-element array, assigns its reference to `ids`, and retrieves a target ID value from the user. Then `main` calls the `findStudent` method, passing copies of the array reference and the particular ID of interest. Using the copy of the passed-in array reference now in its `ids` parameter to access the array created in `main`, the `findStudent` method searches for a given student ID within the student `ids` array. If the student ID is found, it returns the index of the found ID. Otherwise, it returns -1. Note how `findStudent`'s code matches the sequential-search algorithm's logic. In particular, note how `findStudent` implements <*do something and stop the loop*> with a `return i` statement. The `return i` implements "do something" by returning the index of the found student ID. It implements "stop the loop" by returning from the method and terminating the loop simultaneously.



```
/*******************************************************************
* StudentRecords.java
* Dean & Dean
*
* This program searches for a given student id within an array.
*******************************************************************/

import java.util.Scanner;

public class StudentRecords
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    int[] ids = {4142, 3001, 6020};
    int id;       // ID being searched for
    int index;    // index of ID sought or -1 if not found

    System.out.print("Enter 4-digit ID: ");
    id = stdIn.nextInt();
    index = findStudent(ids, id);
    if (index >= 0)
    {
      System.out.println("found at index " + index);
    }
    else
    {
      System.out.println("not found");
    }
  } // end main

  //***************************************************************

  // This method returns index of found id or -1 if not found.

  public static int findStudent(int[] ids, int id)
  {
    for (int i=0; i<ids.length; i++)
    {
      if (ids[i] == id)
      {
        return i;
      }
    } // end for

    return -1;
  } // end findStudent
} // end class StudentRecords
```

Figure S9.10 Program with sequential search method (`findStudent`)



Sample session for Student Records program in Figure S9.10:

```
Enter 4-digit ID: 3001
found at index 1
```

In examining the `findStudent` method, you might be asking yourself "What is the practical use for the returned index?" To do anything with an ID in the `ids` array (add and ID, remove an ID, etc.), you need to know the ID's index. If you don't know the ID's index in advance, the `findStudent` method finds the ID's index for you. Are you still asking yourself "What is the practical use for the returned -1 when the ID is not found?" The -1 can be used by the calling module to check for the case of an invalid student ID.

Binary Search

If you have an array with a large number of array elements, like 100,000, a sequential search typically takes quite a long time. If such an array has to be searched many times, it's often worthwhile to use a binary search. *Binary search* gets its name from the way that it bisects a list of values and narrows its search to just half of the bisected list.

For a binary search to work on an array, the array must be sorted, so that everything is in some kind of alphabetical or numerical order. The next section describes one of the many available sorting methods. This initial sorting takes more time than a single sequential search, but you have to do it only once.

After the array has been sorted, you can use a binary search to find values in the array very quickly—even when the array is extremely long. A sequential search takes an amount of time proportional to the array length. A binary search takes an amount of time proportional to the logarithm of the array length. When an array is very long, the difference between linear and logarithmic is huge. For example, suppose the length is 100,000. It works out that log2(100,000) ≈ 17. Since 17 is about 6,000 times smaller than 100,000, binary search is approximately 6,000 times faster than sequential search for a 100,000-element array.

Note the `binarySearch` method in Figure S9.11. If you'd like to see it used in the context of a complete program, insert it into Figure S9.10's StudentRecords program, and replace this line:

```
index = findStudent(ids, id);
```

with this:

```
index = binarySearch(ids, ids.length, id);
```

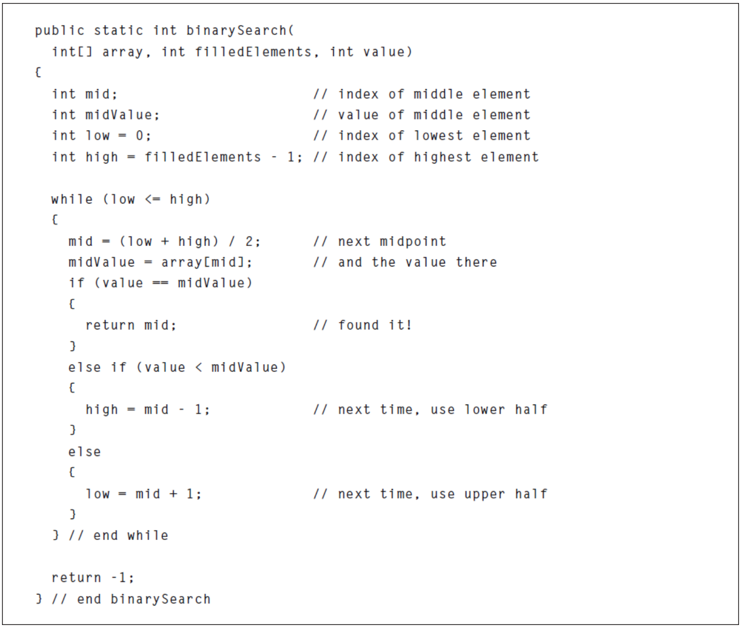
Also, since binary search works only on sorted lists, you should replace this line:

```
int[] ids = {4142, 3001, 6020};
```

with something like this:

```
int[] ids = {3001, 4142,
4890, 6020, 71245, 7265};
```

Before examining the code details in the `binarySearch` method, let's discuss the basic strategy—*divide and conquer*. You first identify the middle element in the sorted array. You then figure out whether the searched-for value goes before or after the middle element. If it goes before the middle element, you narrow the search range to the lower half of the array (the half with the smaller-indexed elements). If, on the other hand, the searched-for value goes after the middle element, you narrow the search range to the upper half of the array. You then repeat the process. In other words, within the narrowed-down half of the array, you identify the middle element, figure out whether the searched-for value goes before or after the middle element, and narrow the search range accordingly. Every time you do this, you cut the problem in half, and this enables you to zero in quickly on the searched-for value—if it's there at all. Splitting the array in half is the "divide" part of "divide and conquer." Finding the searched-for value within one of the halves is the "conquer" part.



```
public static int binarySearch(
  int[] array, int filledElements, int value)
{
  int mid;                        // index of middle element
  int midValue;                   // value of middle element
  int low = 0;                    // index of lowest element
  int high = filledElements - 1;  // index of highest element

  while (low <= high)
  {
    mid = (low + high) / 2;       // next midpoint
    midValue = array[mid];        // and the value there
    if (value == midValue)
    {
      return mid;                 // found it!
    }
    else if (value < midValue)
    {
      high = mid - 1;             // next time, use lower half
    }
    else
    {
      low = mid + 1;              // next time, use upper half
    }
  } // end while

  return -1;
} // end binarySearch
```

Figure S9.11 Method that performs a binary search

Now let's see how the `binarySearch` method implements the divide-and-conquer algorithm. The method declares `mid`, `low`, and `high` variables that keep track of the indexes for the middle element and the two elements at the ends of the array's search range. For an example, see the left drawing in Figure S9.12. Using a `while` loop, the method repeatedly calculates `mid` (the index of the middle element) and checks whether the `mid` element's value is the searched-for value. If the `mid` element's value is the searched-for value, then the method returns the `mid` index. Otherwise, the method narrows the search range to the low half or the high half of the array. Figure S9.12 gives an example of that narrowing process. The method repeats the loop until either the searched-for value is found or the search range shrinks to the point where `low`'s index is greater than `high`'s index.

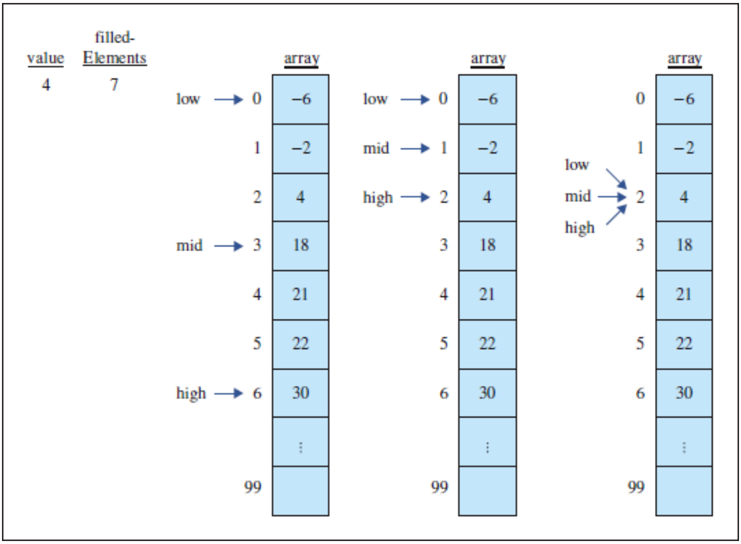

value
filled-Elements
4
7
array
low
mid
high
0
1
2
3
4
5
6
99
−6
−2
4
18
21
22
30
⋮

Figure S9.12 Example execution of Figure S9.11’s `binarySearch` method

S9.8 Sorting an Array

Computers are particularly good at storing large quantities of data and accessing that data quickly. As you learned in the previous section, binary search is an effective technique for finding and accessing data quickly. In order to prepare the data for binary search, the data must be sorted. Sorting data is done not only for binary search purposes. Computers also sort data so that it's easier to display in a user-friendly fashion. If you look at the e-mails in your inbox, aren't they normally sorted by date with the most recent e-mail first? Most e-mail organizers allow you to sort your e-mails using other criteria as well, such as using the "from" person or using the size of the e-mail. In this section, we describe the basics of how sorting is performed. First we present a sorting algorithm, and then we present its implementation in the form of a program that sorts the values in an array.

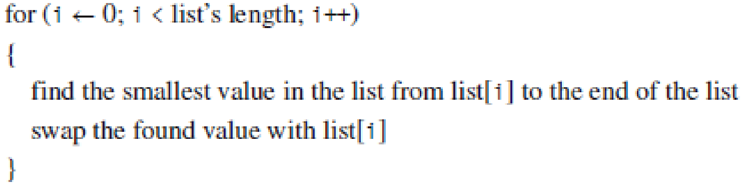
Selection Sort

There are many different sorting algorithms with varying degrees of complexity and efficiency. Frequently, the best way to solve a problem on a computer is the way a human would naturally solve the problem by hand.

To illustrate this idea, we'll show you how to convert one of the common human card-sorting algorithms to a Java sorting program.

If you're sorting cards in a card game, you probably use the *Selection Sort* algorithm. Assume that you're sorting smallest cards first. You search for and select the smallest card and move it to the small-card side of the card group. The small-card side of the card group is where you keep the cards that have been sorted already. You then search for the next smallest card, but in so doing, you look only at cards that are in the unsorted portion of the card group. You move the found card to the second position on the small-card side of the card group. You repeat the search-and-move process until there are no more cards left in the unsorted portion of the card group.

As a first step in implementing the selection sort logic, let's examine a pseudocode solution. Above, we said to “repeat the search-and-move process.” Whenever there’s a repetition, you should think about using a loop. The following algorithm uses a loop for repeating the search-and-move process. Note how `i` keeps track of where the search starts. The first time through the loop, the search starts at the first element (at index 0). The next time, the search starts at the second position. Each time through the loop, you find the smallest value and move it to the sorted portion of the list (the `i` tells you where in the list you want the smallest value to go).



```
for (i ← 0; i < list’s length; i++)
{
   find the smallest value in the list from list[i] to the end of the list
   swap the found value with list[i]
}
```

A picture is worth a thousand words, so we provide a figure (S9.13) that shows the Selection Sort algorithm in action. The five pictures show the different stages of a list being sorted using the Selection Sort algorithm. The list's white portions are unsorted, so the original list at the left is entirely white. The list's blue portions are sorted, so the sorted list at the right is entirely blue. The bidirectional arrows show what happens after a smallest value is found. The smallest value (at the bottom of the bidirectional arrow) gets swapped up to the top of the unsorted portion of the list. For example, in going from the first picture to the second picture, the smallest value, −3, gets swapped up to 5's position at the top of the unsorted portion of the list.

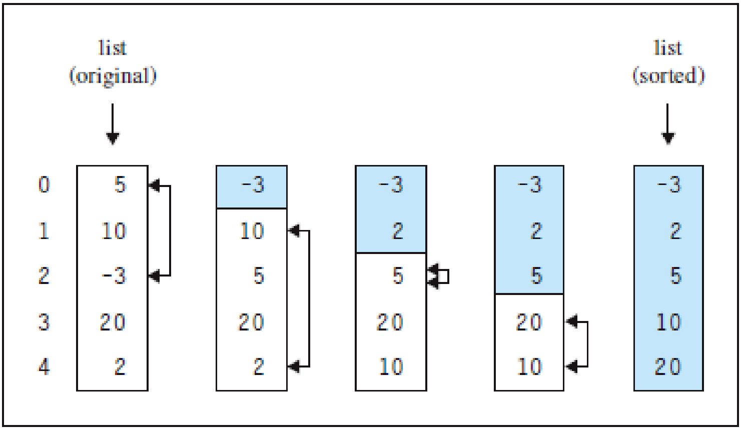
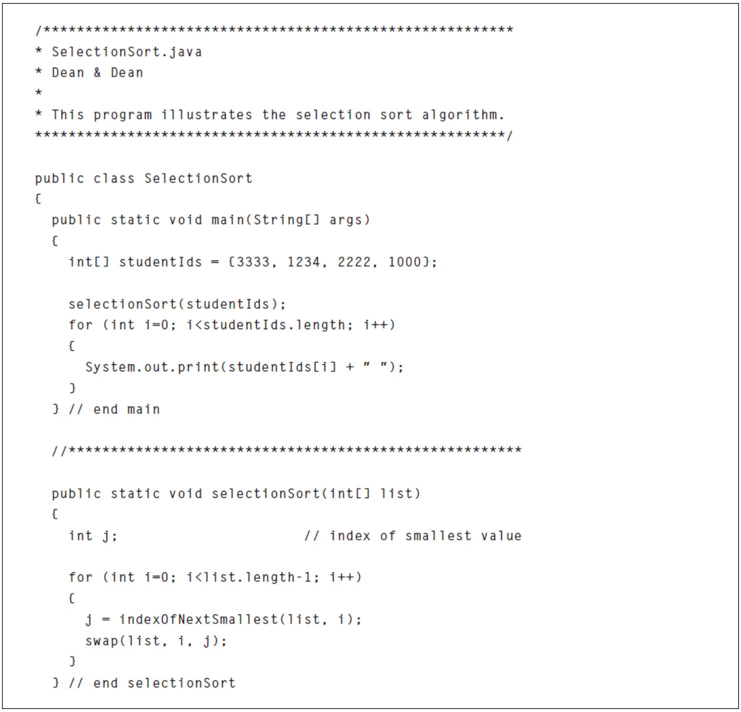

list
(original)
list
(sorted)
0
1
2
3
4
5
10
-3
20
2
-3
10
5
20
2
-3
2
5
20
10
-3
2
5
20
10
-3
2
5
10
20

Figure S9.13 Example execution of the

Selection Sort algorithm

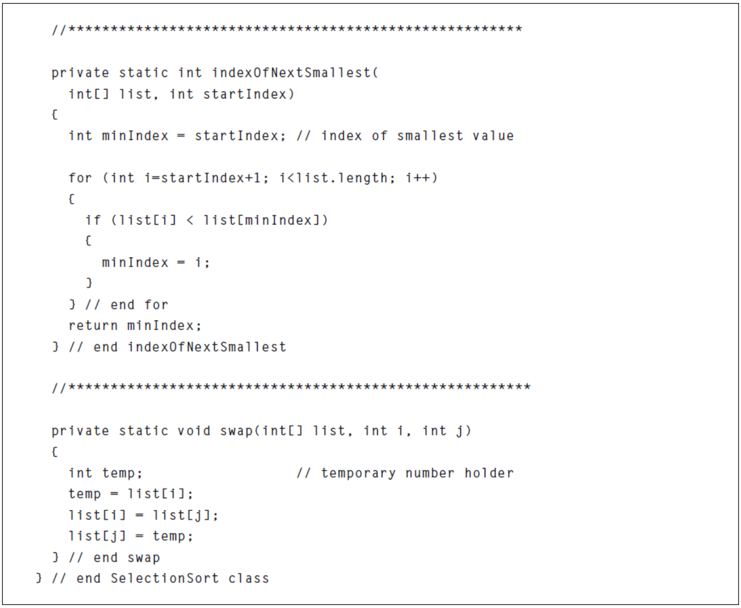
Now let’s implement a Java version of the Selection Sort algorithm. See the `selectionSort` method in Figure S9.14a. Note how its body mimics the pseudocode very closely. The only substantive difference is that the `selectionSort` method’s `for` loop stops iterating one element before the end of the array. That’s because there’s no need to perform a search when you’re at the last element (you already know that the last element is the minimum value for the remainder of the list). We didn’t worry about such efficiency details with the algorithm because algorithms are more about basic logic rather than off-by-one details.



```
/*******************************************************
* SelectionSort.java
* Dean & Dean
*
* This program illustrates the selection sort algorithm.
*******************************************************/

public class SelectionSort
{
  public static void main(String[] args)
  {
    int[] studentIds = {3333, 1234, 2222, 1000};

    selectionSort(studentIds);
    for (int i=0; i<studentIds.length; i++)
    {
      System.out.print(studentIds[i] + " ");
    }
  } // end main

  //*******************************************************

  public static void selectionSort(int[] list)
  {
    int j;                           // index of smallest value

    for (int i=0; i<list.length-1; i++)
    {
      j = indexOfNextSmallest(list, i);
      swap(list, i, j);
    }
  } // end selectionSort
```

Figure S9.14a SelectionSort program that sorts an array of integers—part A

Instead of including the search-for-the-smallest-value code within the `selectionSort` method, we have the `selectionSort` method call an `indexOfNextSmallest` helper method. Instead of including the element-swapping code within the `selectionSort` method, we have the `selectionSort` method call a `swap` helper method. The `indexOfNextSmallest` and `swap` methods appear in Figure S9.14b, which is a continuation of the Selection Sort program . Those methods are `private` because they are helper methods; that is, they help the `selectionSort` method perform its internal tasks.



```
  //*******************************************************

  private static int indexOfNextSmallest(
    int[] list, int startIndex)
  {
    int minIndex = startIndex; // index of smallest value

    for (int i=startIndex+1; i<list.length; i++)
    {
      if (list[i] < list[minIndex])
      {
        minIndex = i;
      }
    } // end for
    return minIndex;
  } // end indexOfNextSmallest

  //********************************************************

  private static void swap(int[] list, int i, int j)
  {
    int temp;                  // temporary number holder
    temp = list[i];
    list[i] = list[j];
    list[j] = temp;
  } // end swap
} // end SelectionSort class
```

Figure S9.14b SelectionSort program that sorts an array of integers—part B

This contains the helper methods called by the selectionSort method in Figure S9.14.

Passing Arrays as Arguments

In Figure S9.14b, notice how the `swap` method receives an array as a parameter (`int[] list`) and implements the swapping operation by updating the array's contents (`list[i] = list[j]; list[j] = temp;`). In order for those updates to do the program any good, they have to impact not only the passed-in array parameter in the `swap` method, but also the corresponding array in the `selectionSort` calling module. Fortunately, such simultaneous updates do occur for method-call arguments that are arrays. Thus, in Figure S9.14a's `selectionSort` method, after the JVM returns from the `swap` method call, the `list` array is indeed updated, with its ith and jth elements having been swapped.

For a full understanding of why arrays exhibit this simultaneous update behavior when they're passed as arguments, you'll need to remember what you learned about reference variables back in Chapter 3. An array variable is a reference variable, and, as such, it holds a reference to an array object. When you pass an array reference to a method, Java's call-by-value scheme causes a copy of that array reference to be passed to the method. The reference copy and the original copy both point to the same array object, so when the array is updated within the method, the array is also updated within the calling module.

Sorting with a Java API Method

When an array has more than about 20 elements, it's better to use an algorithm that's more efficient than the relatively simple Selection Sort algorithm just described. And sure enough, the Java API has a sorting method that uses a more efficient sorting algorithm. It's the `sort` method in the `Arrays` class. Here's skeleton code for how you might use the `sort` method:

```
import java.util.Arrays;

...
  int[] studentIds = {...};

  ...
  Arrays.sort(studentIds);
```

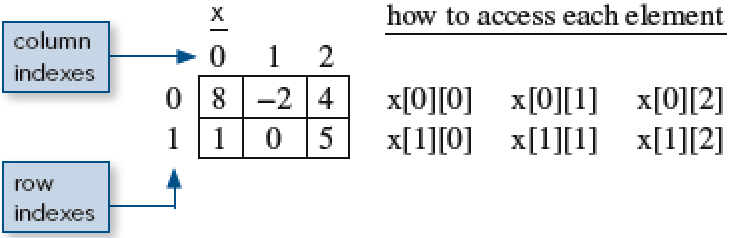
We recommend that you use this API method for heavy-duty sorting. It's an overloaded method, so it also works for arrays of other types of primitive variables.

S9.9 Two-Dimensional Arrays

Arrays are good for grouping related data together. Up to this point, we've grouped the data together using standard one-dimensional arrays. If the related data is organized in a table format, consider using a two-dimensional array. In this section, we describe two-dimensional arrays.

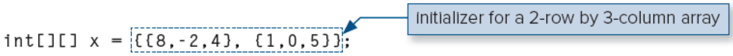
Two-Dimensional Array Syntax

Two-dimensional arrays use the same basic syntax as one-dimensional arrays except for a second pair of square brackets (`[]`). Each pair of square brackets contains one index. According to standard programming practice, the first index identifies the row and the second index identifies the column position within a row. For example, here's a two-row by three-column array named `x`:


column indexes
row indexes
x
0 1 2
0 8 –2 4
1 1 0 5
how to access each element
x[0][0] x[0][1] x[0][2]
x[1][0] x[1][1] x[1][2]

The items at the right, under the "how to access" column heading, show how to access each of the six elements in the array. So to access the value 5, at row index 1 and column index 2, you specify `x[1][2]`.

As with one-dimensional arrays, there are two ways to assign values into a two-dimensional array's elements. You can use an array initializer, where the element assignment is part of the array's declaration. Or you can use standard assignment statements, where the assignment statements are separate from the array's declaration and creation. We'll describe the array initializer technique first. Here's how you can declare the above two-dimensional x array and assign values into its elements, using an array initializer:

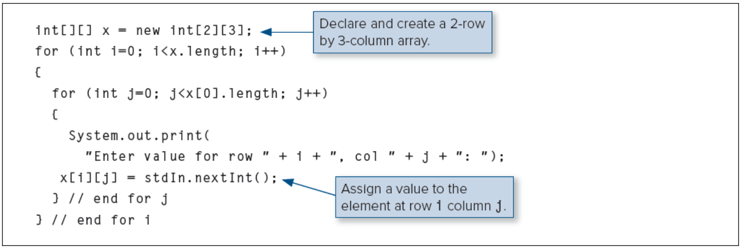


```
int[][] x = {{8,-2,4}, {1,0,5}};
```

Initializer for a 2-row by 3-column array

Note that the array initializer contains two inner groups, where each inner group represents one row. `{8,−2,4}` represents the first row. `{1,0,5}` represents the second row. Note that inner groups are separated with commas, and the entire set of inner groups is surrounded by an enclosing pair of braces.

You can use the array initializer technique only if you know the assigned values when you first declare the array. Otherwise, you need to provide array element assignment statements that are separate from the array's declaration and creation. For example, Figure S9.15's code fragment declares and creates the `x` array in one statement, and assigns values to `x`'s elements in a separate statement, inside nested `for` loops.



```
int[][] x = new int[2][3];
for (int i=0; i<x.length; i++)
{
  for (int j=0; j<x[0].length; j++)
  {
    System.out.print(
      "Enter value for row " + i + ", col " + j + ": ");
   x[i][j] = stdIn.nextInt();
  } // end for j
} // end for i
```

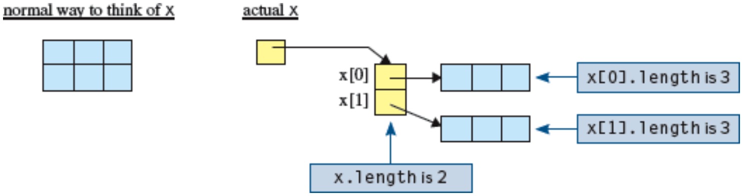
Declare and create a 2-row by 3-column array.

Assign a value to the element at row i column j.

Figure S9.15 Assigning values into a two-dimensional array using nested for loops and the length property

When working with two-dimensional arrays, it’s very common to use nested `for` loops. In Figure S9.15, note the outer `for` loop with index variable `i` and the inner `for` loop with index variable `j`. The outer `for` loop iterates through each row, and the inner `for` loop iterates through each element within a particular row.

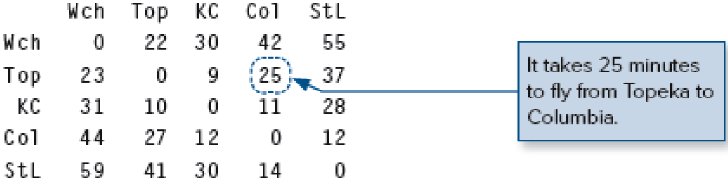
Figure S9.15’s first line declares `x` to be a 2-row by 3-column array with 6 total elements. So you might expect the first `for` loop’s `x.length` property to hold a 6. Not so. Even though it’s normal (and useful) to think of `x` as a rectangular box that holds 6 `int` elements, `x` is actually a reference to a 2-element array and each of the two elements is a reference to its own 3-element array of `int`s. This picture illustrates what we’re talking about:


normal way to think of x
actual x
x[0]
x[1]
x[0].length is 3
x[1].length is 3
x.length is 2

Since `x` is actually a reference to a 2-element array, `x.length` holds the value 2. Or thinking about `x` in the "normal" way (above left picture), `x.length` holds the number of rows in `x`. As you can see above, `x[0]` is a reference to a 3-element array. Thus, `x[0].length` holds the value 3. Or thinking about `x` in the "normal" way (above left picture), `x.length[0]` holds the number of columns in `x`. The point of all this is that the `length` property can be used for iterating through the elements in a two-dimensional array. In Figure S9.15, note how the first loop uses `x.length` to iterate through each row in `x`, and note how the second loop uses `x[0].length` to iterate through each column in `x`.

Example

Let’s put these two-dimensional array concepts into practice by using a two-dimensional array in the context of a complete program. The program, built for a Kansas and Missouri airline company, tells customers when airplanes are expected to arrive at various Kansas and Missouri airports. It uses a two-dimensional array to store flight times between cities, and it displays output like this:

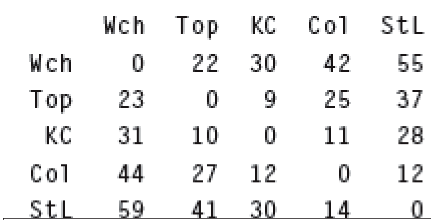


```
      Wch  Top  KC  Col  StL
Wch     0   22  30   42   55
Top    23    0   9   25   37
 KC    31   10   0   11   28
Col    44   27  12    0   12
StL    59   41  30   14    0
```

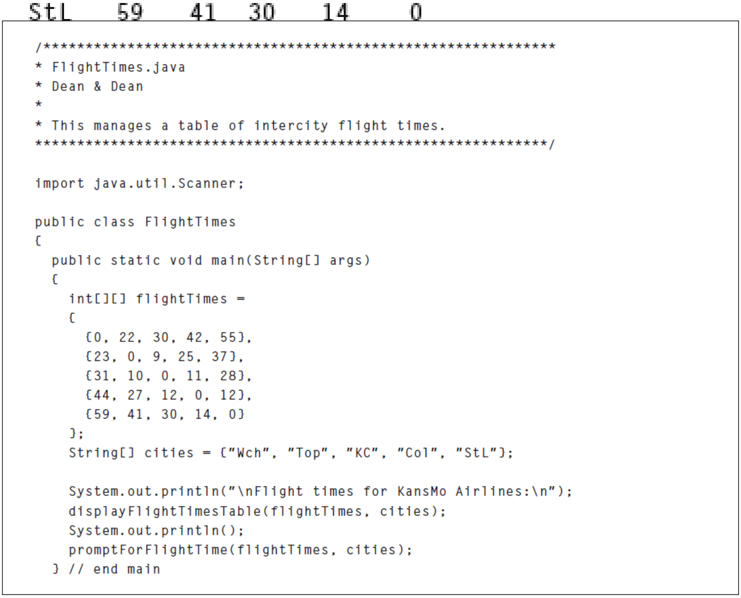
It takes 25 minutes to fly from Topeka to Columbia.

Different rows correspond to different cities of origin. Different columns correspond to different cities of destination. The labels are abbreviations for city names: “Wch” stands for Wichita, Kansas. “Top” stands for Topeka, Kansas. “KC” stands for Kansas City, Missouri. “Col” stands for Columbia, Missouri. “StL” stands for St. Louis, Missouri. Thus, for example, it takes 25 minutes to fly from Topeka to Columbia. How long does it take to go the other way, from Columbia to Topeka? 27 minutes. Columbia to Topeka takes longer because the trip goes east to west, and airplanes have to contend with head winds from North America’s west-to-east jet stream.

Let’s analyze the program by starting with the `main` method in Figure S9.16a. Note how it declares and creates a `flightTimes` table with a two-dimensional array initializer. And note how the initializer puts each table row on a line by itself. That’s not required by the compiler, but it makes for elegant, self-documenting code. It is self-documenting because readers can easily identify each row of table data by looking at a single row of code. After initializing the `flightTimes` table, `main` initializes a one-dimensional array of city names and then calls two methods—`displayFlightTimesTable` and `promptForFlightTime`. We’ll discuss those methods shortly, but to get a feel for the big picture, first take a look at this sample session:

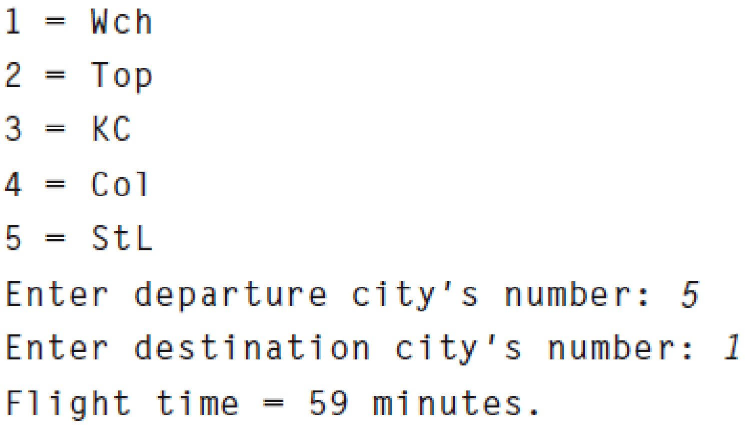


```
Flight times for KansMo Airlines:

      Wch  Top  KC  Col  StL
 Wch    0   22  30   42   55
 Top   23    0   9   25   37
  KC   31   10   0   11   28
 Col   44   27  12    0   12
 StL   59   41  30   14    0
```



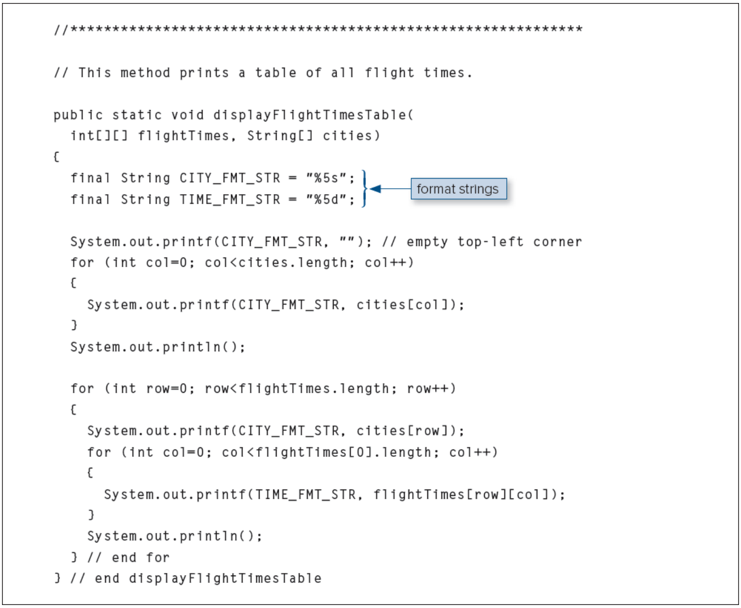
```
/*************************************************************
* FlightTimes.java
* Dean & Dean
*
* This manages a table of intercity flight times.
*************************************************************/

import java.util.Scanner;

public class FlightTimes
{
  public static void main(String[] args)
  {
    int[][] flightTimes =
    {
      {0, 22, 30, 42, 55},
      {23, 0, 9, 25, 37},
      {31, 10, 0, 11, 28},
      {44, 27, 12, 0, 12},
      {59, 41, 30, 14, 0}
    };
    String[] cities = {”Wch”, ”Top”, ”KC”, ”Col”, ”StL”};

    System.out.println(”\nFlight times for KansMo Airlines:\n”);
    displayFlightTimesTable(flightTimes, cities);
    System.out.println();
    promptForFlightTime(flightTimes, cities);
  } // end main
```

Figure S9.16a FlightTimes program—part A



```
1 = Wch
2 = Top
3 = KC
4 = Col
5 = StL
Enter departure city's number: 5
Enter destination city's number: 1
Flight time = 59 minutes.
```

The `displayFlightTimesTable` method appears in Figure S9.16b. It displays the flight times table. In doing so, it employs an interesting formatting technique. First look at the two local named constants, which are separately defined format strings. In Chapter 5 you learned to use literal format strings embedded in strings of text in the arguments of `printf` method calls. But instead of embedding literal format strings, sometimes it's more understandable to declare format strings separately as named constants. If you go back and count the spaces in the six-column table of flight times, you'll see that each column is exactly 5 spaces wide. So the labels at the top of the columns and the numbers in the columns must both be formatted to use exactly 5 spaces. Thus, the format string for the labels (`CITY_FMT_STR`) should be `"%5s"`, and the format string for the integer entries (`TIME_FMT_STR`) should be "%5d". Using named constants for format strings allows each format string to be used in many places, and it makes it easy and safe to alter them at any later time—just change the values assigned to the named constants at the beginning of the method.

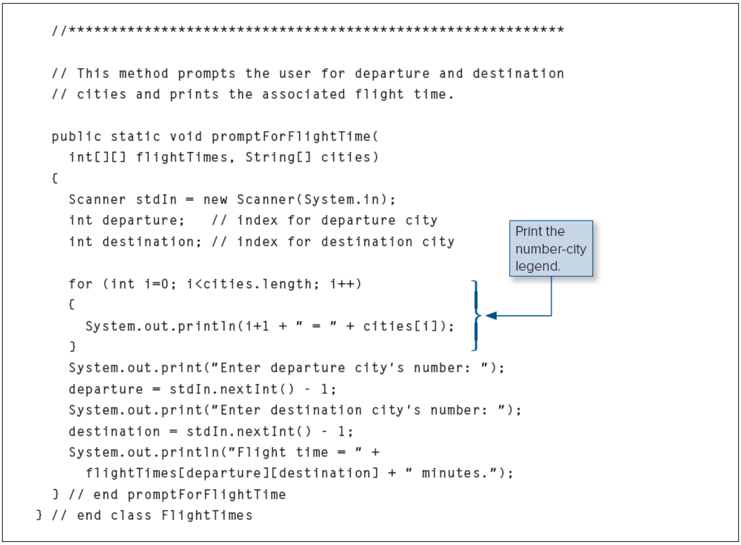


```
//***************************************************************

// This method prints a table of all flight times.

public static void displayFlightTimesTable(
  int[][] flightTimes, String[] cities)
{
  final String CITY_FMT_STR = "%5s";
  final String TIME_FMT_STR = "%5d";

  System.out.printf(CITY_FMT_STR, ""); // empty top-left corner
  for (int col=0; col<cities.length; col++)
  {
    System.out.printf(CITY_FMT_STR, cities[col]);
  }
  System.out.println();

  for (int row=0; row<flightTimes.length; row++)
  {
    System.out.printf(CITY_FMT_STR, cities[row]);
    for (int col=0; col<flightTimes[0].length; col++)
    {
      System.out.printf(TIME_FMT_STR, flightTimes[row][col]);
    }
    System.out.println();
  } // end for
} // end displayFlightTimesTable
```

Figure S9.16b FlightTimes program—part B

In the `displayFlightTimesTable` method, note the three `for` loop headers. They all use the `length` property for their termination condition. Since `length` holds 5, the program would run correctly if you replaced the length termination conditions with hardcoded 5’s. But don’t do it. Using the `length` property makes the implementation more *scalable*. Scalable means it’s easy to change the amount of data that the program uses. For example, in the FlightTimes program, using a `cities.length` loop termination condition means that if you change the number of cities in the program, the program will still work properly.

The `promptForFlightTime` method appears in Figure S9.16c. This method prompts the user for a departure city and a destination city and prints the flight time for that flight. More specifically, it prints a legend of numbers and their associated city names (1 = Wichita, 2 = Topeka, etc.), it prompts the user to enter numbers for the departure and destination cities, and it prints the flight time between the specified cities. Note how user-entered city numbers start with 1 rather than 0 (1 = Wichita). That makes the program more user-friendly because people usually prefer to start counting at one rather than zero. Internally, the program stores city names in an array. Since all arrays start with a 0 index, the program has to translate between user-entered city numbers (which start at 1) and city array indexes (which start at 0). Note how that's done with "+ 1" and "− 1" in the `promptForFlightTime` method.



```
  //*************************************************************

  // This method prompts the user for departure and destination
  // cities and prints the associated flight time.

  public static void promptForFlightTime(
    int[][] flightTimes, String[] cities)
  {
    Scanner stdIn = new Scanner(System.in);
    int departure;   // index for departure city
    int destination; // index for destination city

    for (int i=0; i<cities.length; i++)
    {
      System.out.println(i+1 + " = " + cities[i]);
    }
    System.out.print("Enter departure city's number: ");
    departure = stdIn.nextInt() - 1;
    System.out.print("Enter destination city's number: ");
    destination = stdIn.nextInt() - 1;
    System.out.println("Flight time = " +
      flightTimes[departure][destination] + " minutes.");
  } // end promptForFlightTime
} // end class FlightTimes
```

Print the number-city legend.

Figure S9.16c FlightTimes program—part C

Multidimensional Arrays

Arrays may have more than two

dimensions. Arrays with three or more dimensions use the same basic syntax except they have additional `[]`'s. The first pair of brackets corresponds to the largest scale, and each subsequent pair of brackets corresponds to progressively smaller levels of scale. For example, suppose the Missouri-Kansas airline company decides to go "green" and expands its fleet with new airplanes powered by photovoltaic solar cells on their wings and airplanes that burn hydrogen generated by wind turbines. The new airplanes have different flight times than the original jet-fuel airplanes. Thus, they need their own flight-times tables. The solution is to create a three-dimensional array where the first dimension specifies the largest level of scale—the airplane's type: 0 for the jet-fuel airplanes, 1 for the solar-powered airplanes, and 2 for the hydrogen-powered airplanes. Here's how to declare the new three-dimensional `flightTimes` array variable:

```
int[][][] flightTimes;
```

GUI Programming Basics

Objectives

- Understand the event-driven programming paradigm. In particular, understand what it means to fire an event, and understand the terms *listener* and *event handler*.
- Use the JFrame class to implement window functionality.

- Create and use JLabel, JTextField, and JButton components.
- Implement a listener for the JTextField and JButton components.
- Understand what an interface is and implement the ActionListener interface.
- Understand what an inner class is and implement a listener as an inner class.
- Know the difference between an anonymous inner class and a standard inner class.
- Create and use JOptionPane dialog boxes.
- Be able to distinguish between multiple events.
- Describe the primary GUI packages.

- Describe the difference between lightweight and heavyweight components.
- Learn how to use mouse listeners with images.

Outline

17.1 Introduction •••

17.2 Event-Driven Programming Basics •••

17.3 A Simple Window Program •••

17.4 JFrame Class •••

17.5 Java Components •••

17.6 JLabel Component •••

17.7 JTextField Component •••

17.8 Greeting Program •••

17.9 Component Listeners •••

17.10 Inner Classes •••

17.11 Anonymous Inner Classes •••

17.12 JButton Component •••

17.13 Dialog Boxes and the JOptionPane Class •••

17.14 Distinguishing Between Multiple Events •••

17.15 Using getActionCommand to Distinguish Between Multiple Events •••

17.16 Color •••

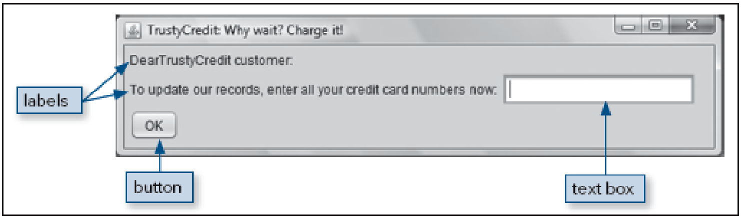
17.17 How GUI Classes Are Grouped Together •••

17.18 Mouse Listeners and Images (Optional) •••

17.1 Introduction

Hopefully, you’ve been on the edge of your seat in reading the prior chapters. If not, be prepared to be on the edge of your seat now. It’s time for the really good stuff: *graphical user interface* (*GUI*) programming.

You’ve probably heard the term *GUI*, and you probably know that it’s pronounced “gooey.” But do GUI’s three words, *graphical user interface*, make sense? “Graphical” refers to pictures, “user” refers to a person, and “interface” refers to communication. Thus, GUI programming employs pictures—like windows, labels, text boxes, buttons, and so on—to communicate with users. For example, Figure 17.1 shows a window with two labels, a text box, and a button. We’ll describe windows, labels, text boxes, and buttons in detail later on.



Dear TrustyCredit customer:

To update our records, enter all your credit card numbers now:

Figure 17.1 Example window that uses two labels, a text box, and a button

In the old days, program interfaces consisted of just text. Programs would prompt the user with a text question, and users would respond with a text answer. That's what we've been using for all of our programs so far. Text input/output (I/O) works well in many situations, but you can't get around the fact that some people consider text display to be boring. Many of today's users expect programs to be livelier. They expect windows, buttons, colors, and so on for input and output. They expect GUI.

Although companies still write many text-based programs for internal use, they normally write GUI-based programs for programs that are to be used externally. It’s important for external programs to be GUI based because external programs go to customers, and customers typically won’t buy programs unless they are GUI based. So if you want to write programs that people will buy, you’d better learn GUI programming.

We start this chapter with an overview of basic GUI concepts and terminology. We then move on to a bare-bones program where we introduce basic GUI syntax. We next cover listeners, inner classes, and - several rudimentary *GUI components,* which are objects that sit inside a window, including JLabel, JTextField, and JButton. We describe the JOptionPane class (for generating a dialog box) and the Color class (for generating a color). Finally, we present mouse listeners and images. Section 17.18 shows how to use a mouse to drag an image around in a window.

You may have noticed optional GUI-track sections at the end of about half of the prior chapters. The GUI material in this chapter and the next is different from the GUI material in the earlier chapters, and it does not depend on the earlier chapters' GUI material. So if you skipped the earlier GUI material, no worries.

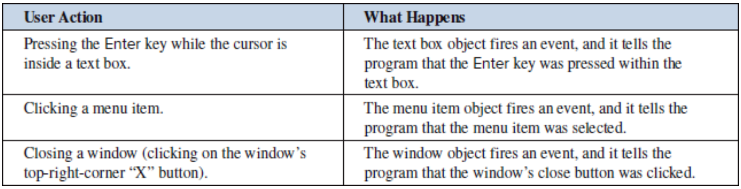
To understand this chapter, you need to be familiar with object-oriented programming, arrays, inheritance, and exception handling. As such, you need to have read up through Chapter 15. This chapter does not depend on material covered in Chapter 16.

17.2 Event-Driven Programming Basics

GUI programs usually use *event-driven programming* techniques. The basic idea behind event-driven programming is that the program waits for events to occur and the program responds to events if and when they occur.

Terminology

So what is an event? An *event* is a message that tells the program that something has happened. For example, if the user clicks a button, then an event is generated, and it tells the program that a particular button was clicked. More formally, when the user clicks a button, we say that the button object *fires an event.* Note these additional event examples:



User Action	What Happens
Pressing the Enter key while the cursor is inside a text box.	The text box object fires an event, and it tells the program that the Enter key was pressed within the text box.
Clicking a menu item.	The menu item object fires an event, and it tells the program that the menu item was selected.
Closing a window (clicking on the window's top-right-corner "X" button).	The window object fires an event, and it tells the program that the window's close button was clicked.

If an event is fired, and you want your program to handle the fired event, then you need to create a *listener* for the event. For example, if you want your program to do something when the user clicks a particular button, you need to create a listener for the button. For now, think of a listener as an ear. If an event is fired and there's no ear listening to it, then the fired event is never "heard" and there's no response to it. On the other hand, if there is an ear listening to a fired event, then the ear "hears" the event and the program then responds to the fired event. The way the program responds is by executing a chunk of code known as an *event handler*. See Figure 17.2. It depicts a button being pressed (see the mouse pointer), an event being fired (see the sound waves), a listener hearing the event (see the ear), and an event handler being executed (see the arrow going down the event-handler code). This system of using listeners for event handling is known as the *event-delegation model*—event handling is "delegated" to a particular listener.

The Event-Driven Programming

Framework

Based on the above description, event-driven programming may feel like an altogether new type of programming. Particularly the part about firing an event and listening for a fired event. Many people are fine with the idea of event-driven programming being a new type of programming. But the truth of the matter is that it's really just object-oriented programming with window dressing. Make that lots of window dressing. Oracle provides an extensive collection of GUI classes that, together, form a framework on which to build GUI applications. That framework is comprised of classes, methods, inheritance, and so on. In other words, it's comprised of OOP components. As a programmer, you don't have to understand all the details of how the framework works; you just have to understand it well enough to use it. For example, you have to know how to plug in your event handlers properly. Figure 17.3 provides a high-level, graphic illustration of what we're talking about.

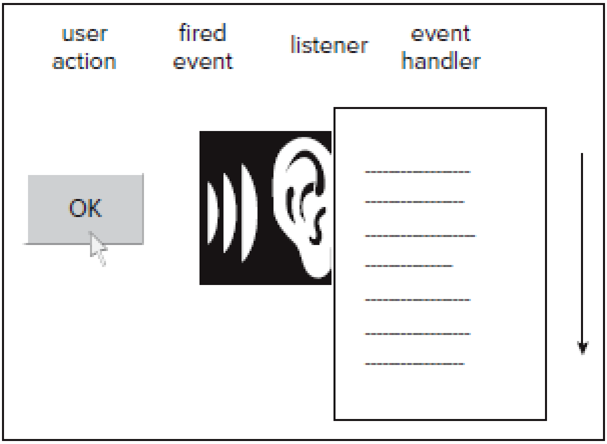

user action
fired event
listener
event handler
OK

Figure 17.2 What happens when a button is pressed

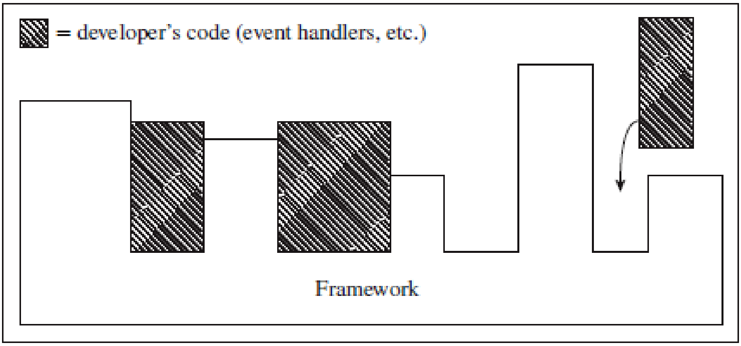

▨ = developer's code (event handlers, etc.)
Framework

Figure 17.3 Event-driven programming framework

Why did the Java language designers bother to provide the event-driven programming framework? It satisfies the goal of getting maximum benefit from minimum code. With the help of the framework, Java programmers can get a GUI program up and running with a relatively small amount of effort. Initially, the effort might not seem so small, but when you consider all that the GUI program does (automatic event firing, listening for fired events, and so on), you'll find that your return on investment is quite good.

17.3 A Simple Window Program

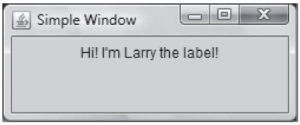
OK. Enough talk about concepts. Time to roll up your sleeves and get your hands dirty with some code. To get a feel for the big picture, let's start with a simple GUI program and discuss the GUI commands at a high level. Later, we'll cover the GUI commands in greater detail.

In Figure 17.4, we present a SimpleWindow program that displays a line of text inside a window. Note the two import statements at the top of the program. They import the javax.swing and java.awt packages. In writing GUI programs, you'll use many of Java's prebuilt GUI classes from Oracle's API library. To use the prebuilt GUI classes, you'll need to import them into your GUI programs. You could import the classes individually, but there's a better way. Recall that a package is a collection of prebuilt classes. Because most of the critical prebuilt GUI classes come from the javax.swing and java.awt packages, import those two packages and you'll import most of the critical prebuilt GUI classes. Get used to importing those two packages in every one of your GUI programs. Recall that to import a package, you need to use an asterisk, like this:

```
/*************************************************************
* SimpleWindow.java
* Dean & Dean
*
* This program displays a label in a window.
*************************************************************/

import javax.swing.*; // for JFrame, JLabel
import java.awt.*;    // for FlowLayout

public class SimpleWindow extends JFrame
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 100;

  //*******************************************************

  public SimpleWindow()
  {
    setTitle("Simple Window");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end SimpleWindow constructor

  //*******************************************************

  private void createContents()
  {
    JLabel label = new JLabel("Hi! I'm Larry the label!");
    add(label);
  } // end createContents

  //*******************************************************

  public static void main(String[] args)
  {
    new SimpleWindow();
  } // end main
} // end class SimpleWindow
```

This adds the label to the window.

This instantiates an anonymous window object.



Hi! I'm Larry the label!

Figure 17.4 SimpleWindow program and its output

import javax.swing.*;

The * is a wildcard, and it allows you to import all the classes within a particular package.

In SimpleWindow's class heading, note the extends JFrame clause. The JFrame class is part of the GUI framework mentioned previously. The JFrame class provides standard Windows features such as a title bar, a minimize button, and so on. Below the class heading, note the WIDTH and HEIGHT named constants. They're used by the setSize method call to specify the dimensions of the window.

Let's now examine the main method. GUI programs typically create a window with GUI components, and then they just sit around waiting for the user to do something like click a button, select a menu option, and so on. Thus, main is very short—it just instantiates the window and that's it. In this simple example, we don't even bother to assign the instantiated window object to a reference variable. Review: What do you call an object that isn't stored in a reference variable? An anonymous object.

In performing the anonymous-object instantiation, main calls the SimpleWindow constructor. The SimpleWindow constructor (1) calls setTitle to assign the window’s title, (2) calls setSize to assign the window’s size, (3) calls setLayout to assign the window’s layout scheme, and (4) calls setDefaultCloseOperation to enable the *close-window button* (the “X” in the top-right corner) to work properly.

In the interest of modularization, the SimpleWindow constructor then calls a helper method, createContents, to create the components that go inside the window. The createContents method contains only two lines. With only two lines, there’s really no need for a helper method, but we want you to form good habits. For this trivial example, there’s only one component and there’s no event handler for the component. Thus, two lines are all that's needed. But normal GUI programs have multiple components and multiple event handlers. For that, quite a few lines are needed. If you stick those lines in the constructor, you'd have a long constructor. Better to break things up and stick them in a helper method.

The createContents method instantiates a JLabel component and then calls the add method to add the JLabel component to the window. A JLabel component is the simplest type of GUI component. It's a piece of text that the user can read but cannot change.

After executing createContents, the JVM returns to the SimpleWindow constructor. The SimpleWindow constructor then calls setVisible to make the window visible.

17.4 JFrame Class

In the previous section, we introduced you to the JFrame class. In this section, we describe the JFrame class in more depth. More specifically, we cover its characteristics and its methods.

JFrame Basics

These days, most purchasable software is windows-based. When you load such software, you'll see a window and that window will have a title bar, a border, a minimize button, a close-window button, the ability to resize the window, and so on. You could implement all those features from scratch in your own classes, but why "reinvent the wheel"? The JFrame class implements the standard windows features that you've come to know and love. To get all that cool windows functionality for free, just implement your classes by extending the JFrame class. What a deal!

The JFrame class should be the

superclass for most of your GUI application windows, so a -programmer- defined window will normally have extends JFrame in its class heading. For the extends JFrame to work, you must import the JFrame class or import JFrame's package, javax.swing. As explained above, it's common to import the javax.swing package for all GUI programs.

The JFrame class is called a *container* because it contains components (like labels, buttons, menus, and so on). It inherits the ability to contain components from its superclass, the Container class.

JFrame Methods

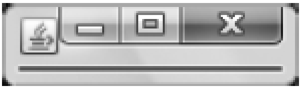
By extending the JFrame class, you automatically get the standard windows functionality mentioned above. In addition, you inherit a host of windows-related methods. In the SimpleWindow program, we use these inherited methods—setTitle, setSize, setLayout, setDefaultCloseOperation, add, and setVisible. The setLayout and setDefaultCloseOperation methods come directly from the JFrame class. The other methods come from ancestors of the JFrame class—setTitle from the Frame class, add from the Container class, setSize and setVisible from the Component class.

The setTitle method displays a specified string in the current window's title bar. If setTitle is not called, then the window's title bar is empty.

The setSize method assigns the width and height of the current window. See Figure 17.4 and note how the SimpleWindow program assigns the width to 250 and the height to 100. The width and height values are specified in terms of *pixels*. A pixel is a computer monitor's smallest displayable unit, and it displays as a dot on the screen. If you call setSize with a width of 250 and a height of 100, then your window will consist of 100 rows where each row contains 250 pixels. Each pixel displays with a certain color. The pixels form a picture by having different colors for the different pixels. For example, the window depicted in Figure 17.4 might contain blue pixels on the perimeter (for the window's border), and black pixels in the center (for the window's message).

To give you perspective on how big a 250-by-100 pixel window is, you need to know the dimensions, in pixels, of an entire computer screen. The dimensions of a computer screen are referred to as the screen's *resolution*. Resolution settings are adjustable. Two common resolution settings are 800-by-600 and 1024-by-768. The 800-by-600 setting displays 600 rows where each row contains 800 pixels.

If you forget to call the setSize method, your window will be really small. It will display only the ⚠ beginning of the title and the three standard window-adjustment buttons—minimize, maximize, and close-window. It won't display the window's contents unless you manually resize the window. Here's what the SimpleWindow program displays if you omit the setSize method call:



The setLayout method assigns a specified *layout manager* to the current window. The layout manager is prebuilt Java API software that determines the positioning of components. In the SimpleWindow program's setLayout call, we specify the FlowLayout manager, and the FlowLayout manager causes components to be positioned in the top-center position. The FlowLayout class is defined in the java.awt package, so don't forget to import that package. In the next chapter, we describe the FlowLayout manager and other layout managers in more detail. We're using the FlowLayout manager (as opposed to other layout managers) in this chapter because the FlowLayout manager is the easiest to use, and we're trying to keep things simple for now.

⚠ By default, a program's close-window button (the X in the top-right corner) doesn't work very nicely. When the user clicks it, the window closes, but the program still runs in the background. To remedy this situation, call setDefaultCloseOperation(EXIT_ON_CLOSE). Then when the user clicks the close-window button, the window closes and the program terminates. Having a closed program run in the background is usually unnoticeable, and that's why many programmers have a hard time remembering to call setDefaultCloseOperation(EXIT_ON_CLOSE). Nonetheless, you should try to remember to call it. If you forget to call it, and a user's computer has limited memory and there are many programs running in the background, the computer's performance will degrade.

The add method adds a specified component to the current window. Once the component is added, it stays with the window for the life of the program. We mention this so that you're comfortable using a local variable declaration for a component. In the following example, although label is defined locally within createContents, the instantiated JLabel component stays with the window after createContents finishes:

```
private void createContents()
{
  JLabel label = new JLabel(”Hi! I’m Larry the label!”);
  add(label);
} // end createContents
```

Windows are invisible by default. To make a window and its contents visible, add the ⚠components to the window and then call setVisible(true). Do it in that order—add components first, then call setVisible. Otherwise, the added components won’t display. To make a window invisible, call setVisible(false).

The JFrame class contains many additional methods, too many to mention here. If you’ve got some time on your hands, we encourage you to find out what’s available by looking up the JFrame class on Oracle’s Java API website—http://download.oracle.com/javase/7/docs/api/

17.5 Java Components

Now let's consider the objects that sit inside a window—the components. Here are some examples of Java components:

- JLabel, JTextField, JButton
- JTextArea, JCheckBox, JRadioButton, JComboBox
- JMenuBar, JMenu, JMenuItem

These aren't all of the Java components, just some of the more commonly used ones. We'll describe the first three components in this chapter and the other components in the next chapter.

All of these component classes are in the javax.swing package, so you must import that package to use them. But remember that you're already importing the javax.swing package to access the JFrame class. There's no need to import it twice.

Component classes typically are derived from the JComponent class, which supports many useful inheritable features. Along with many other methods, the JComponent class contains methods that handle these component features:

- foreground and background colors
- text font
- border appearance
- tool tips
- focus

For detailed information on these features, look up the JComponent class on Oracle's Java API website.

17.6 JLabel Component

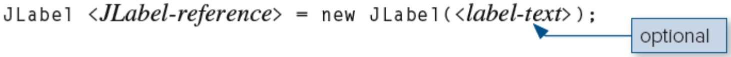
User Interface

The JLabel component doesn't do much. It simply displays a specified single line of text. It's considered to be a read-only component because the user can read it, but the user cannot interact with it.

Normally, the JLabel component displays a single line of text, not multiple lines. If you want to display multiple lines, use the JTextArea component, which is covered in the next chapter.

Implementation

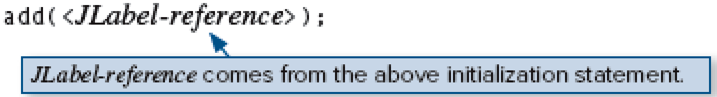
To create a JLabel object, call the JLabel constructor like this:



```
JLabel <JLabel-reference> = new JLabel(<label-text>);
```

The *label-text* is the text that appears in the JLabel component. If the label-text argument contains a newline character, \n, it's ignored (remember that the JLabel component displays only a single line of text). If the label-text argument is omitted, then the JLabel component displays nothing. Why instantiate an empty label? So you can fill it in later on with text that's dependent on some condition.

To add a JLabel object to your JFrame window, use this syntax:



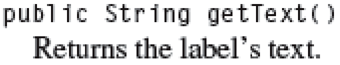
```
add(<JLabel-reference>);
```

JLabel-reference comes from the above initialization statement.

The JLabel class needs the javax.swing package, but that should be available already because it's needed for the JFrame class.

Methods

The JLabel class, like all the GUI component classes, has quite a few methods. We'll just mention two of them—the getText and setText accessor and mutator methods. Here are their API headings and descriptions:



```
public String getText()
```

Returns the label's text.

public String getText()

Returns the label's text.

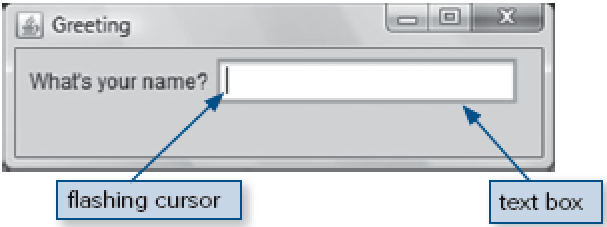
public void setText(String text)

Assigns the label's text. Note that the programmer can update the label's text even though the user cannot.

17.7 JTextField Component

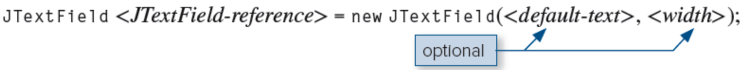
User Interface

The JTextField component displays a rectangle and allows the user to enter text into the rectangle. Here's an example:


Greeting
What's your name?
flashing cursor
text box

Implementation

To create a JTextField object, call the JTextField constructor like this:



`JTextField` *<JTextField-reference>* `= new JTextField(`*<default-text>*, *<width>*`);`

The *default-text* is the text that appears in the text box by default. The *width* is the number of characters that can display in the text box at one time. If the user enters more characters than can display at one time, then the leftmost characters scroll off the display. If the default-text argument is omitted, then the empty string is used as the default. If the width argument is omitted, then the box's width is slightly greater than the width of the default text.

To add a JTextField object to your JFrame window, use this syntax:

add(<*JTextField-reference*>);

The JTextField class needs the javax.swing package, but that should be available already because it's needed for the JFrame class.

Methods

The JTextField class has quite a few methods. Here are API headings and descriptions for some of the more useful ones:

public String getText()

Returns the text box's contents.

public void setText(String text)

Assigns the text box's contents.

public void setEditable(boolean flag)

Makes the text box editable or non-editable.

public void setVisible(boolean flag)

Makes the text box visible or invisible.

public void addActionListener(ActionListener listener)

Adds a listener to the text box.

Text boxes are editable by default, which means users can type inside them. If you want to prevent users from editing a text box, call setEditable with an argument value of false. Calling setEditable(false) prevents users from updating a text box, but it does not prevent programmers from updating a text box. Programmers can call the setText method regardless of whether the text box is editable or non-editable.

Components are visible by default, but there are some instances where you might want to call setVisible(false) and make a component disappear. After you calculate a result, you might want just the result to appear without the clutter of other components. When a component is made to disappear, its space is automatically reclaimed by the window so other components can use it.

When a JTextField component calls

addActionListener, the JVM attaches a listener object to the text box, and that enables the program to respond to the user pressing Enter within the text box. We'll cover listeners in more detail soon enough, but first we're going to step through an example program that puts into practice what you've learned so far. . . .

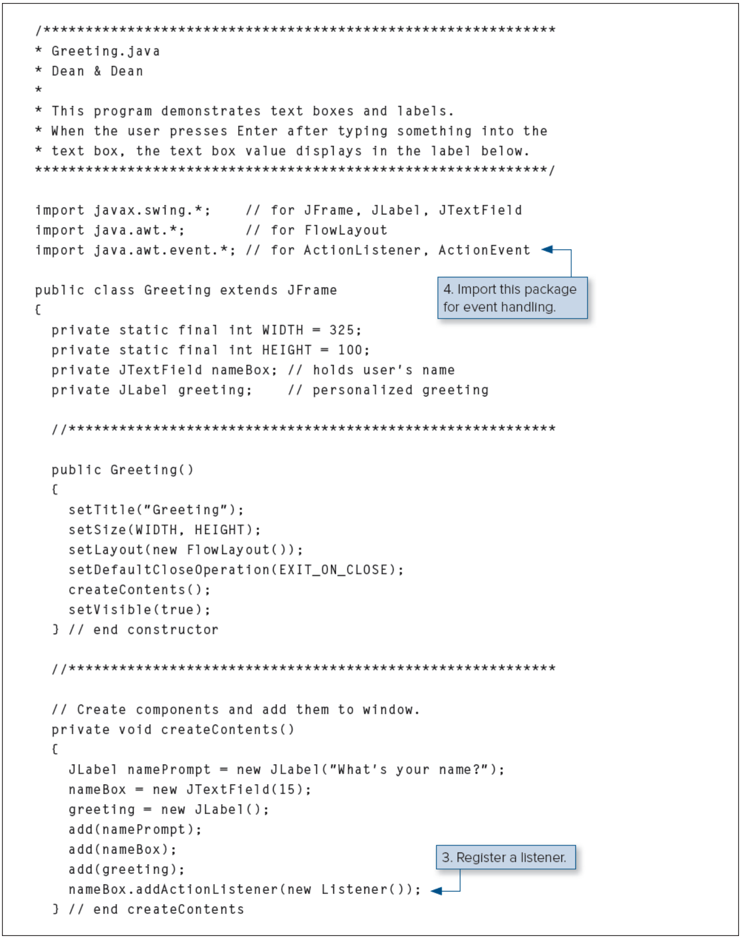
17.8 Greeting Program

In Figures 17.5a and 17.5b, we present a Greeting program that displays a personalized greeting. It reads the user's name from a text box (a JTextField component) and displays the entered name in a label (a JLabel component).

Most of the code in the Greeting program should look familiar because it closely parallels the code in the SimpleWindow program. For example, notice the short main method with the anonymous object instantiation. Also notice how the constructor contains calls to setTitle, setSize, setLayout, setDefaultCloseOperation, and setVisible. Finally, note the createContents helper method that creates the components and adds them to the window. Now let's focus on what's new about the Greeting program—a text box and an event handler.

The Greeting program uses a text box called nameBox to store the user's name. Note how the createContents method instantiates nameBox with a width of 15. Note how the createContents method calls the add method to add nameBox to the window. That code is straightforward. But something that's not so straightforward is nameBox's declaration. It's declared as an instance variable at the top of the class. Why an instance variable instead of a createContents local variable? Aren't local variables preferred? Yes, but in this case, we need to access nameBox not only in createContents, but also in the actionPerformed event handler (which we'll get to next). It's possible to use a local variable within createContents and still access it from the event handler, but that's a bit of a pain.[1] For now, we'll keep things simple and declare the nameBox as an instance variable. We'll do the same with the greeting label, because we need to access it in createContents and also in the actionPerformed event handler.

The Greeting program's actionPerformed event handler specifies what happens when the user presses Enter within the text box. Note that the actionPerformed method is inside our Listener class. We cover listeners and event-handler mechanics in the next section.

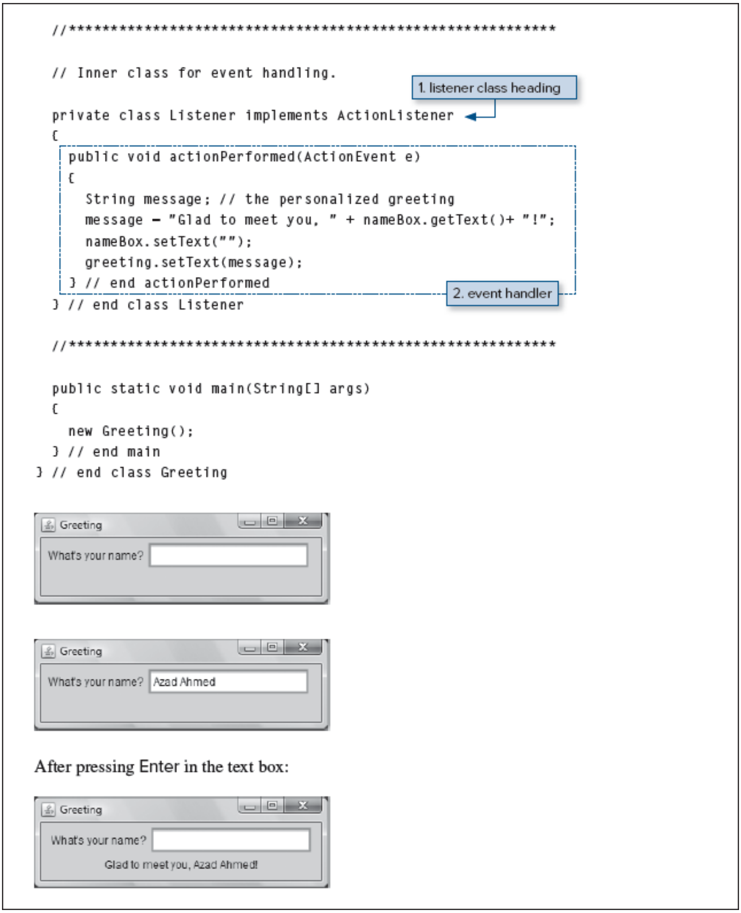


```
/*************************************************************
* Greeting.java
* Dean & Dean
*
* This program demonstrates text boxes and labels.
* When the user presses Enter after typing something into the
* text box, the text box value displays in the label below.
*************************************************************/

import javax.swing.*;    // for JFrame, JLabel, JTextField
import java.awt.*;       // for FlowLayout
import java.awt.event.*; // for ActionListener, ActionEvent

public class Greeting extends JFrame
{
  private static final int WIDTH = 325;
  private static final int HEIGHT = 100;
  private JTextField nameBox; // holds user's name
  private JLabel greeting;    // personalized greeting

  //***********************************************************

  public Greeting()
  {
    setTitle("Greeting");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end constructor

  //***********************************************************

  // Create components and add them to window.
  private void createContents()
  {
    JLabel namePrompt = new JLabel("What's your name?");
    nameBox = new JTextField(15);
    greeting = new JLabel();
    add(namePrompt);
    add(nameBox);
    add(greeting);
    nameBox.addActionListener(new Listener());
  } // end createContents
```

4. Import this package for event handling.

3. Register a listener.

Figure 17.5a Greeting program—part A



```
  //************************************************************

  // Inner class for event handling.

  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      String message; // the personalized greeting
      message = "Glad to meet you, " + nameBox.getText()+ "!";
      nameBox.setText("");
      greeting.setText(message);
    } // end actionPerformed
  } // end class Listener

  //************************************************************

  public static void main(String[] args)
  {
    new Greeting();
  } // end main
} // end class Greeting
```

What's your name?

What's your name? Azad Ahmed

After pressing Enter in the text box:

What's your name?

Glad to meet you, Azad Ahmed!

Figure 17.5b Greeting program—part B, and its associated output

17.9 Component Listeners

When the user interacts with a component (e.g., when the user clicks a button or presses Enter while in a text box), the component fires an event. If the component has a listener attached to it, the fired event is "heard" by the listener. Consequently, the listener handles the event by executing its actionPerformed method. In this section, you'll learn how to make all that work by creating a listener and an associated -actionPerformed method.

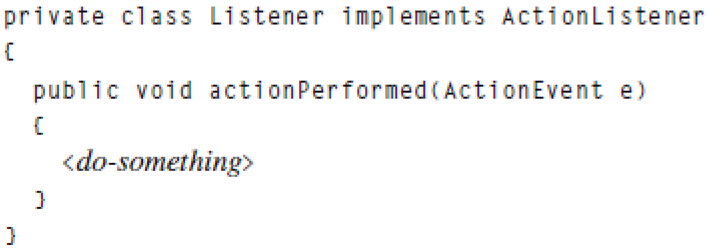
How to Implement a Listener

Below, we show the steps needed to implement a listener for a text box. These steps correspond to the numbered callouts in Figures 17.5a and 17.5b:

1. Define a class with an implements ActionListener clause appended to the right of the class's heading. To see an example, look at callout 1 in Figure 17.5b. The implements ActionListener clause means that

the class is an implementation of the ActionListener interface. We discuss interfaces in the next subsection.

2. Include an actionPerformed event handler method in your listener's class. Here's a skeleton of an actionPerformed method inside a listener class:



```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    <do-something>
  }
}
```

Even if your actionPerformed method doesn't use the ActionEvent parameter (e, above), you still must include that parameter in the method heading to make your method conform to the requirements of a listener.

To see an example of a complete actionPerformed method, look at callout 2 in Figure 17.5b. It refers to a listener class that's named Listener.

Listener is not a reserved word—it's just a good descriptive name we picked for the listener class in the Greeting program.

3. **3.** *Register* your listener class. More specifically, that means adding your listener class to a text box component by calling the addActionListener method. Here's the syntax:

   <*text-box-component*>.addActionListener(new <*listener-class*>());

   To see an example, look at callout 3 in Figure 17.5a.

   The point of the registration process is so your text box can find a listener when an *enter event* is fired. An enter event is fired whenever the user presses Enter from within the text box.

   Registering a listener is like registering your car. When you register your car, nothing much happens at that point. But later, when some event occurs, your car registration comes into play. What event would cause your car registration to be used? If you get caught speeding, the police can use your registration number as part of a traffic citation. If you get into a wreck, your insurance company can use your registration number to raise your insurance rates.

4. Import the java.awt.event package. Event handling requires the use of the ActionListener interface and the ActionEvent class. Those entities are in the java.awt.event package, so that package must be imported for event handling to work. To see the import statements within a complete program, look at callout 4 in Figure 17.5a.

The ActionListener Interface

In the Greeting program, we specified implements ActionListener in the listener's class heading. ActionListener is an *interface*. You might recall interfaces from Chapter 14. An interface is somewhat like a class in that it contains variables and methods. But unlike a class, an interface's variables must be constants (implemented with the final modifier), its methods must be empty (implemented as method headings), and an interface cannot be instantiated. If a programmer uses an interface to derive a new class, the compiler requires the new class to implement methods for all of the interface's method headings.

So what's the point of having an interface with all empty methods? The answer is that it can be used as a template or pattern when creating a class that falls into a certain category.

More specifically, what's the point of the ActionListener interface? Because all action-event listeners must implement it, it means that all action-event listeners will be similar and therefore understandable. It means that all action-event listeners will implement the ActionListener's one method, the actionPerformed method. And in implementing that method, they'll be forced to use this prescribed heading:

```
public void actionPerformed(ActionEvent e)
```

By using the prescribed heading, it ensures that fired action events will be received properly by the listener.

17.10 Inner Classes

Here's a reprint of the Greeting program, in skeleton form:

```
public class Greeting extends JFrame
{
  ...
  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      String message; // the personalized greeting
      message = "Glad to meet you, " + nameBox.getText();
      nameBox.setText("");
      greeting.setText(message);
    } // end actionPerformed
  } // end class Listener
  ...
} // end class Greeting
```

Do you notice anything odd about the position of the Listener class in the Greeting program? See how the Listener class is indented and how its closing brace is before the Greeting class's closing brace? The Listener class is inside the Greeting class!

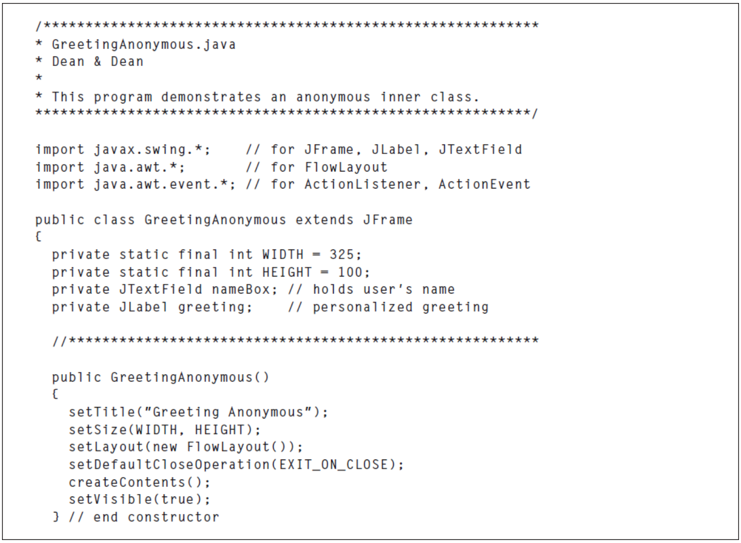
If a class is limited in its scope such that it is needed by only one other class, you should define the class as an *inner class* (a class inside another class). Because a listener is usually limited to listening to just one class, listeners are usually implemented as inner classes.

It's not required by the compiler, but inner classes should normally be private. Why? Because the main point of using an inner class is to further the goal of encapsulation and using private means the outside world won't be able to access the inner class. Note the private modifier in the above Listener class heading.

Besides furthering the goal of encapsulation, there's another reason to use an inner class as opposed to a *top-level* class (*top-level class* is the formal term for a regular class—a class not defined inside another class). An inner class can access its enclosing class's instance variables directly. Because listeners normally need to access their enclosing class's instance variables, this is an important benefit.

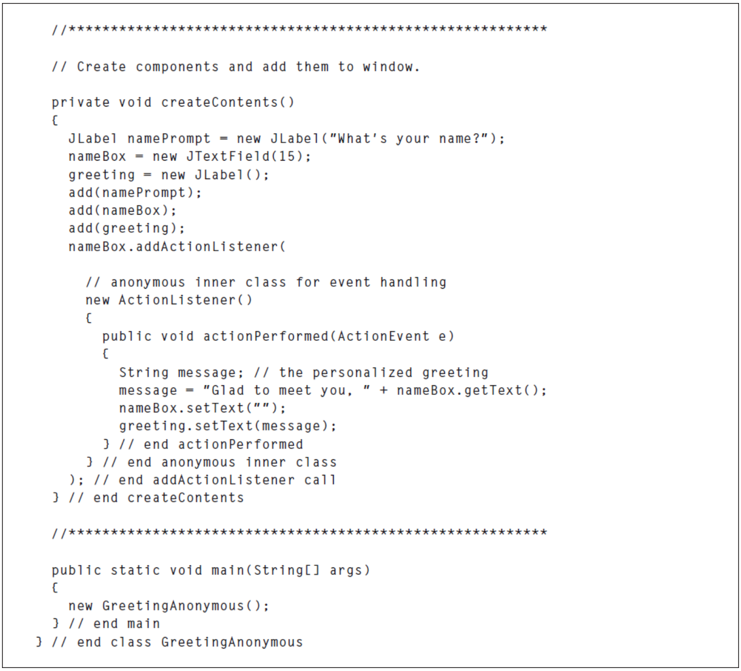
17.11 Anonymous Inner Classes

Take a look at the GreetingAnonymous program in Figures 17.6a and 17.6b. It’s virtually identical to the previous Greeting program. Can you identify the difference between the GreetingAnonymous program and the Greeting program?



```
/***********************************************************
* GreetingAnonymous.java
* Dean & Dean
*
* This program demonstrates an anonymous inner class.
***********************************************************/

import javax.swing.*;    // for JFrame, JLabel, JTextField
import java.awt.*;       // for FlowLayout
import java.awt.event.*; // for ActionListener, ActionEvent

public class GreetingAnonymous extends JFrame
{
  private static final int WIDTH = 325;
  private static final int HEIGHT = 100;
  private JTextField nameBox; // holds user’s name
  private JLabel greeting;    // personalized greeting

  //*********************************************************

  public GreetingAnonymous()
  {
    setTitle(“Greeting Anonymous”);
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end constructor
```

Figure 17.6a GreetingAnonymous program, which has an anonymous inner class—part A



```
  //***********************************************************

  // Create components and add them to window.

  private void createContents()
  {
    JLabel namePrompt = new JLabel("What's your name?");
    nameBox = new JTextField(15);
    greeting = new JLabel();
    add(namePrompt);
    add(nameBox);
    add(greeting);
    nameBox.addActionListener(

      // anonymous inner class for event handling
      new ActionListener()
      {
        public void actionPerformed(ActionEvent e)
        {
          String message; // the personalized greeting
          message = "Glad to meet you, " + nameBox.getText();
          nameBox.setText("");
          greeting.setText(message);
        } // end actionPerformed
      } // end anonymous inner class
    ); // end addActionListener call
  } // end createContents

  //***********************************************************

  public static void main(String[] args)
  {
    new GreetingAnonymous();
  } // end main
} // end class GreetingAnonymous
```

Figure 17.6b GreetingAnonymous program, which has an anonymous inner class—part B

In the Greeting program, we implemented a listener class named Listener, using this code:

```
private class Listener implements ActionListener
{
```

That code is omitted in the GreetingAnonymous program—there's no class named Listener. But we still need a listener object so that the text box's enter event is detected and acted upon. This time, instead of declaring a listener class with a name (e.g., Listener), we implement a listener class anonymously (without a name).

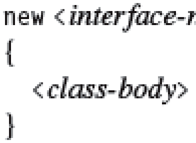
We've discussed anonymous objects previously. That's where you instantiate an object without storing its reference in a variable. In our previous Greeting program, we instantiated an anonymous Listener object with this line:

```
nameBox.addActionListener(new
Listener());
```

The point of using an anonymous object is to avoid cluttering the code with a variable name when an object needs to be used only one time. The same idea can be applied to classes. The point of using an *anonymous inner class* is to avoid cluttering up the code with a class name when a class needs to be used only one time. For example, if a particular listener class listens to just one object, then the listener class needs to be used only one time as part of an addActionListener method call. Therefore, to unclutter your code, you may want to use an anonymous inner class for the listener.

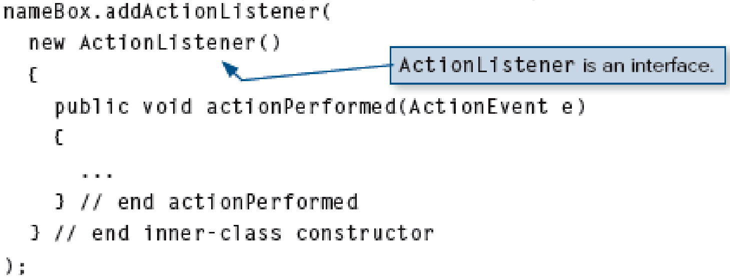
Using an anonymous inner class is not a compiler requirement. It's an elegance issue. In industry, you'll find some people who say anonymous inner classes are elegant and you'll find other people who say anonymous inner classes are confusing. Do as you see fit. Better yet, do as your teacher sees fit.

Below, we show the syntax for an anonymous inner class. Naturally, there's no class name. But there is an interface name. So anonymous inner classes aren't built from scratch; they're built with the help of an interface.[2] Note the new operator. Formally speaking, the new operator isn't part of the anonymous inner class. But practically speaking, because there's no point in having an anonymous inner class without instantiating it, you can think of the new operator as being part of the anonymous inner class syntax.



```
new <interface-name> ()
{
  <class-body>
}
```

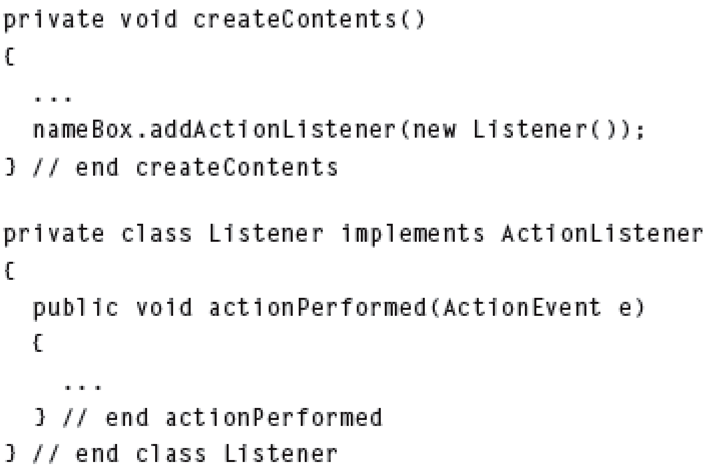
Here's an example of an anonymous inner class, taken from the GreetingAnonymous program:



```
nameBox.addActionListener(
  new ActionListener()
  {
    public void actionPerformed(ActionEvent e)
    {
      ...
    } // end actionPerformed
  } // end inner-class constructor
);
```

`ActionListener` is an interface.

For comparison purposes, here's an example of a named (non-anonymous) inner class. It's taken from the Greeting program:



```
private void createContents()
{
  ...
  nameBox.addActionListener(new Listener());
} // end createContents

private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    ...
  } // end actionPerformed
} // end class Listener
```

There are only two syntactic differences between the two code fragments—the addActionListener call and the listener class heading. There are no semantic differences between the two code fragments, so the Greeting program and the GreetingAnonymous program behave the same.

17.12 JButton Component

It's now time to learn another GUI component—a button component.

User Interface

If you press a button on an electronic device, something usually happens. For example, if you press the power button on a TV's remote control, the TV turns on or off. Likewise, if you press/click a GUI *button* component, something usually happens. For example, in Figure 17.1's TrustyCredit window, if you click the OK button, the entered credit card numbers get processed by the TrustyCredit company.

Implementation

To create a button component, call the JButton constructor like this:

```
JButton helloButton = new JButton("Press me");
```

button label's text

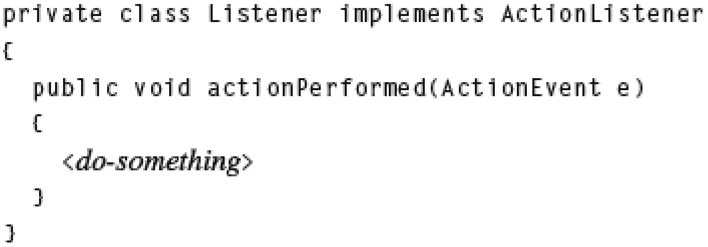
When this button is displayed, it says "Press me" in the center of the button. The label argument is optional. If it's omitted, the label gets the empty string by default and the button displays with a blank face (no writing or icons on it).

After you have created the

helloButton, add it to your window, like this:

add(helloButton);

To make the button useful, you’ll need to implement a listener. As with the text box listeners, button listeners must implement the ActionListener interface. The ActionListener interface dictates that you must have an actionPerformed event handler method. The code skeleton looks like this:



```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    <do-something>
  }
}
```

We’re using private instead of public for the listener class because a listener normally is implemented as an inner class, and inner classes are normally private. We’re using a named inner class instead of an anonymous inner class because named inner classes are slightly more flexible. They allow you to create a listener that's used on more than one component. We'll provide an example in an upcoming program.

To register the above listener with our helloButton component, do this:

```
helloButton.addActionListener(new Listener());
```

The JButton class needs the javax.swing package, but that should be available already because it's needed for the JFrame class. The ActionListener interface and the ActionEvent class need the java.awt.event package, so import that package.

Methods

Here are API headings and descriptions for some of the more useful JButton methods:

```
public String getText()
```

Returns the button's label.

public void setText(String text)
Assigns the button's label.
public void setVisible(boolean flag)

Makes the button visible or invisible.
public void addActionListener(ActionListener listener)

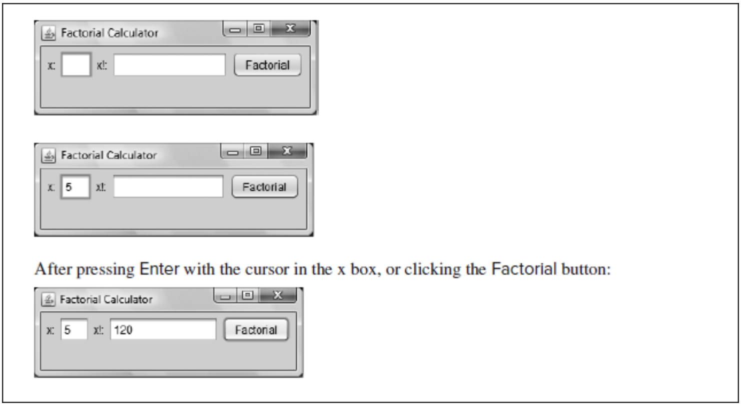
Adds a listener to the button. The listener "listens" for the button being clicked.

FactorialButton Program

It's time to put all this JButton syntax into practice by showing you how it's used within a complete program. We've written a FactorialButton program that uses a JButton component to calculate the factorial for a user-entered number.[3] To give you a better idea of how the program operates, see the sample session in

Figure 17.7.

Figures 17.8a and 17.8b contain the FactorialButton program listing. Most of the code should already make sense because the program's structure parallels the structure in our previous GUI programs. We'll skip the more familiar code and focus on the more difficult code.

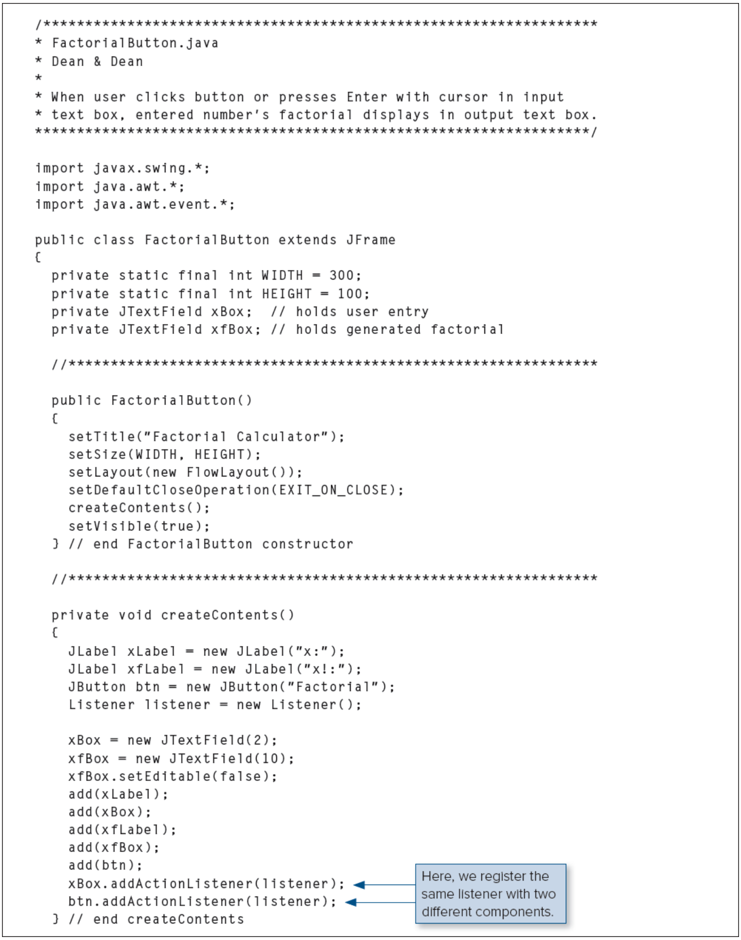


x: 5

After pressing Enter with the cursor in the x box, or clicking the Factorial button:

x: 5 x!: 120

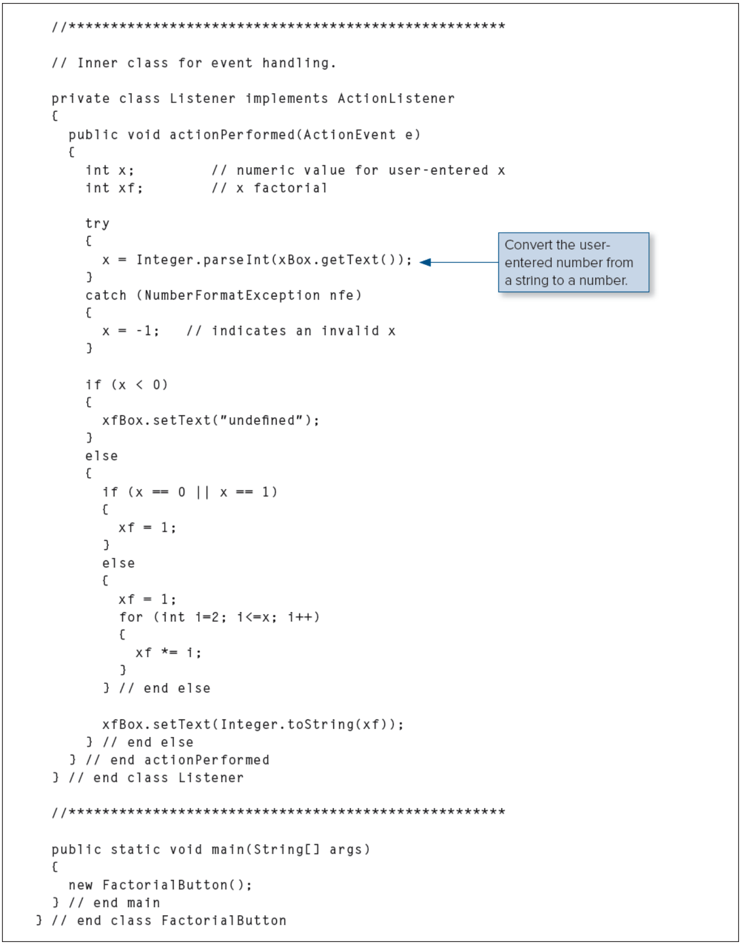
Figure 17.7 Sample session for the FactorialButton program



```
/*******************************************************************
* FactorialButton.java
* Dean & Dean
*
* When user clicks button or presses Enter with cursor in input
* text box, entered number's factorial displays in output text box.
*******************************************************************/

import javax.swing.*;
import java.awt.*;
import java.awt.event.*;

public class FactorialButton extends JFrame
{
  private static final int WIDTH = 300;
  private static final int HEIGHT = 100;
  private JTextField xBox;  // holds user entry
  private JTextField xfBox; // holds generated factorial

  //****************************************************************

  public FactorialButton()
  {
    setTitle("Factorial Calculator");
    setSize(WIDTH, HEIGHT);
    setLayout(new FlowLayout());
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end FactorialButton constructor

  //****************************************************************

  private void createContents()
  {
    JLabel xLabel = new JLabel("x:");
    JLabel xfLabel = new JLabel("x!:");
    JButton btn = new JButton("Factorial");
    Listener listener = new Listener();

    xBox = new JTextField(2);
    xfBox = new JTextField(10);
    xfBox.setEditable(false);
    add(xLabel);
    add(xBox);
    add(xfLabel);
    add(xfBox);
    add(btn);
    xBox.addActionListener(listener);
    btn.addActionListener(listener);
  } // end createContents
```

Here, we register the same listener with two different components.

Figure 17.8a FactorialButton program—part A



```
  //*****************************************************

  // Inner class for event handling.

  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      int x;         // numeric value for user-entered x
      int xf;        // x factorial

      try
      {
        x = Integer.parseInt(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;   // indicates an invalid x
      }

      if (x < 0)
      {
        xfBox.setText(”undefined”);
      }
      else
      {
        if (x == 0 || x == 1)
        {
          xf = 1;
        }
        else
        {
          xf = 1;
          for (int i=2; i<=x; i++)
          {
            xf *= i;
          }
        } // end else

        xfBox.setText(Integer.toString(xf));
      } // end else
    } // end actionPerformed
  } // end class Listener

  //*****************************************************

  public static void main(String[] args)
  {
    new FactorialButton();
  } // end main
} // end class FactorialButton
```

Convert the user-entered number from a string to a number.

Figure 17.8b FactorialButton program—part B

We declare most of our GUI variables locally within createContents, but we declare the two text box components as instance variables at the top of the program. Why the difference? As discussed earlier, normally you should declare components as local variables to help with encapsulation. But if a component is needed in createContents and also in an event handler, it's fine to declare it as an instance variable where it can be shared more easily. In the FactorialButton program, we declare the two text boxes as instance variables because we need to use them in createContents and also in the actionPerformed event handler.

Note this line from the createContents method:

```
xfBox.setEditable(false);
```

This causes the factorial text box, xfBox, to be non-editable (i.e., the user won't be able to update the text box). That should make sense because xfBox holds the factorial, and it's up to the program (not the user) to generate the factorial. Note in Figure 17.7 that the factorial text box is grayed out. You get that visual cue free of charge whenever you call setEditable(false) from a text box component. Cool!

Again from the createContents method:

```
Listener listener = new Listener();
...
xBox.addActionListener(listener);
btn.addActionListener(listener);
```

Note that we're registering the same listener with two different components. By doing this, we give the user two ways to trigger a response. The user can press Enter when the cursor is in the input text box (xBox) or the user can click on the button (btn). Either way causes the listener to react. Whenever you register the same listener with two different components, you need to have a name for the listener. That's why we use a named inner class for this program (an anonymous inner class wouldn't work).

Figure 17.8b's actionPerformed method is chock full of interesting code. The first thing you see is perhaps the most important—the Integer.parseInt method call. If you ever need to read numbers or display numbers in a GUI program, you have to use string versions of the numbers. Thus, to read a number from the input text box, we first read it in as a string, and then we convert the string to a number. To accomplish this, we read the string using xBox.getText(), and we convert it to a number using Integer.parseInt.

Ideally, you should always check user input to make sure it's valid. In the actionPerformed method, we check for two types of invalid input—a non-integer input and a negative number input. Those inputs are invalid because the factorial is mathematically undefined for those cases. The negative number case is easier, so we'll start with it. Note this code in the middle of the ActionPerformed method:

```
if (x < 0)
{
xfBox.setText("undefined");
}
```

x is the user’s entry after it’s been converted to an integer. If x is negative, the program displays -undefined in the xfBox component.

Now for the non-integer input case. Note this code near the top of the ActionPerformed method:

```
try
{
x = Integer.parseInt(xBox.getText());
}
catch (NumberFormatException nfe)
{
x = -1; // indicates an invalid x
}
```

The Integer.parseInt method attempts

to convert xBox's user-entered value to an integer. If xBox's user-entered value is a non-integer, then parseInt throws a NumberFormatException. To handle that possibility, we put the Integer.parseInt method call inside a try block, and we include an associated catch block. If parseInt throws an exception, we want to display undefined in the xfBox component. To do that, we could call xfBox.setText("undefined") in the catch block, but then we'd have redundant code—xfBox.setText("undefined") in the catch block and also in the subsequent if statement. To avoid code redundancy and its inherent maintenance problems, we assign −1 to x in the catch block. That causes the subsequent if statement to be true and that in turn causes xfBox.setText("undefined") to be called.

Write compact code.

After validating the input, the actionPerformed method calculates the factorial. It first takes care of the special case when x equals 0 or 1. It then takes care of the x ≥ 2 case by using a for loop. Study the code. It works fine, but do you see a way to make it more compact? You can omit the block of code that starts with if (x = = 0 || x = = 1) because that case is handled by the else block. More specifically, you can delete the six lines above the second xf = 1; line.

17.13 Dialog Boxes and the JOptionPane Class

A *dialog box*—often referred to simply as a *dialog*—is a specialized type of window. The primary difference between a dialog box and a standard window is that a dialog box is more constrained in terms of what it can do.

While a standard window usually remains on the user's screen for quite a while (often for the duration of the program) and performs many tasks, a dialog box remains on the screen only long enough to perform one specific task. While a standard window is highly customizable, a dialog box typically is locked into one particular format.

User Interface

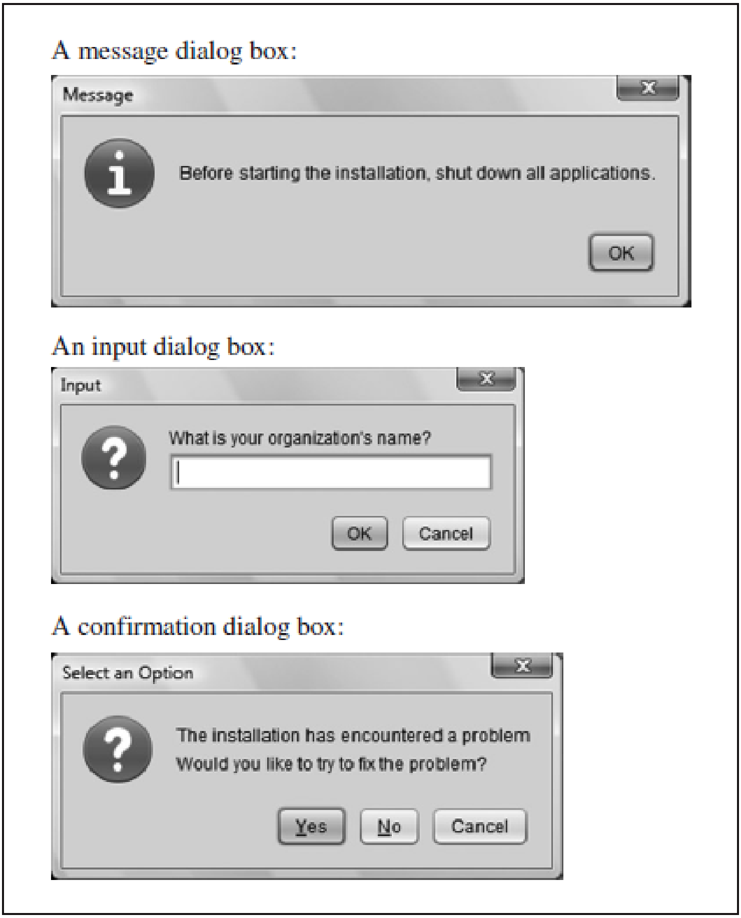
There are three types of JOptionPane dialogs—a *message dialog,* an *input dialog,* and a *confirmation dialog*. Each type performs one specific task. The message dialog displays output. The input dialog displays a question and an input field. The confirmation dialog displays a yes/no question and yes/no/cancel button options. See what the different types look like in Figure 17.9. The optional GUI section at the end of Chapter 3 used the first two of these types, the message dialog and the input dialog.

Implementation

To create a message dialog box, call the showMessageDialog method like this:

JOptionPane.showMessageDialog(<*contai*
<*message*>);

Because showMessageDialog is a class method in the JOptionPane class, you need to prefix the showMessageDialog call with "JOptionPane dot". Remember—call instance methods using "<*reference- variable*> dot" syntax, and call class methods using "<*class-name*> dot" syntax.


A message dialog box:
Message
OK
An input dialog box:
Input
OK
Cancel
A confirmation dialog box:
Select an Option
Yes
No
Cancel

Before starting the installation, shut down all applications.

What is your organization's name?

The installation has encountered a problem
Would you like to try to fix the problem?

Figure 17.9 Three types of JOptionPane dialog boxes

To create an input dialog box, call the showInputDialog method like this:

String input = JOptionPane.showInputDialog(<*container*>
<*message*>);

The returned String value, input, is the value the user enters in the text field. To create a confirmation dialog box, call the showConfirmDialog method like this:

```
int selection =
JOptionPane.showConfirmDialog(<container
<message>);
```

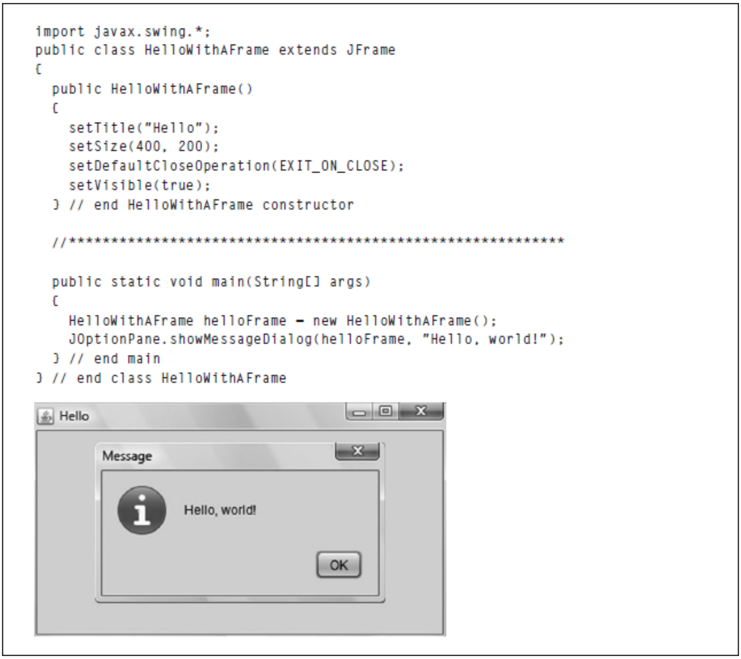
The returned int value, selection, is one of JOptionPane’s static constants. If the user clicks Yes, the returned value is JOptionPane.YES_OPTION. If the user clicks No, the returned value is JOptionPane.NO_OPTION. If the user clicks Cancel, the returned value is JOptionPane.CANCEL_OPTION. You can use these JOptionPane constants as case values in a switch statement.

As indicated in Java API documentation, the JOptionPane class also provides other methods and options. Sometimes, as in the GUI program at the end of Chapter 3, we just want the dialog box to pop out all by itself—not within any container. For that, we make the <*container*> argument null. At other times, we want the dialog to appear inside a larger window. For that, we make the <*container*> argument be a JFrame object. In all cases, the <*message*> argument is the dialog box's text.

Note the showMessageDialog call in Figure 17.10's HelloWithAFrame program. This call uses helloFrame for showMessageDialog's container argument. What type of container is it? As the code indicates, helloFrame is an instance of the HelloWithAFrame class, and the HelloWithAFrame class extends the JFrame container. Therefore, by inheritance, helloFrame is a JFrame container. And consequently, the dialog box displays in the center of the program's JFrame container. Verify this by looking at Figure 17.10's output.

Suppose you don’t want to bother with centering the dialog within a particular container. In that case, use null for showMessageDialog’s container argument. That causes the dialog to display in the center of the screen. For example, this code fragment generates a screen-centered dialog:

```
JOptionPane.showMessageDialog(
null, "Before starting the installation, \n" +
"shut down all applications.");
```



```
import javax.swing.*;
public class HelloWithAFrame extends JFrame
{
  public HelloWithAFrame()
  {
    setTitle("Hello");
    setSize(400, 200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setVisible(true);
  } // end HelloWithAFrame constructor

  //**********************************************************

  public static void main(String[] args)
  {
    HelloWithAFrame helloFrame - new HelloWithAFrame();
    JOptionPane.showMessageDialog(helloFrame, "Hello, world!");
  } // end main
} // end class HelloWithAFrame
```

Hello, world!

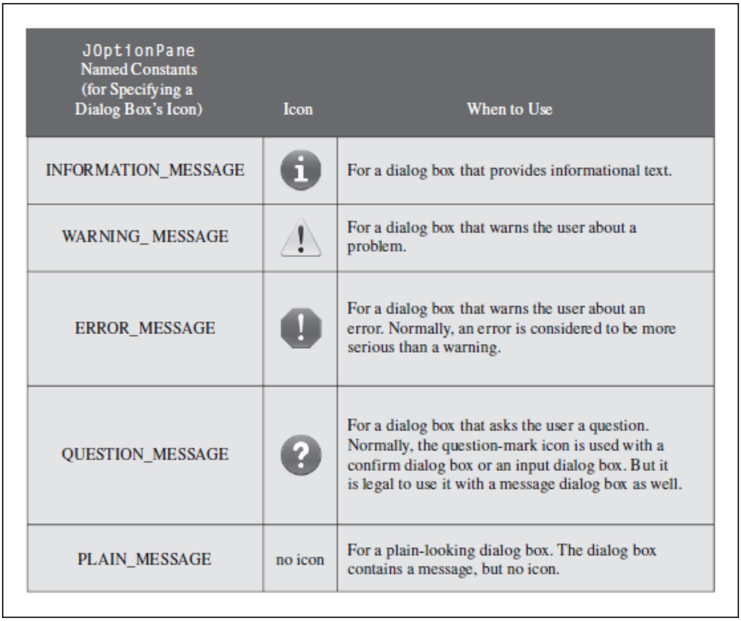
Figure 17.10 HelloWithAFrame program and its output

By the way, it's very common to use null for showMessageDialog's container argument, probably more common than using a non-null value.

The JOptionPane class needs the javax.swing package. If you've imported the javax.swing package for the JFrame class already, there's no need to import it again.

Method Details

In Figure 17.10, note the message dialog's title-bar message—it's "Message." Kinda boring, eh? To liven things up, add a third argument to the showMessageDialog call that specifies the dialog's title. Also in Figure 17.10, note the message dialog's icon—it's an i inside a circle. That's the default icon. To specify an icon explicitly, add a fourth argument to the showMessageDialog call that specifies one of the named constants in Figure 17.11.



`JOptionPane` Named Constants (for Specifying a Dialog Box's Icon)	Icon	When to Use
INFORMATION_MESSAGE	ℹ	For a dialog box that provides informational text.
WARNING_MESSAGE	⚠	For a dialog box that warns the user about a problem.
ERROR_MESSAGE	❗	For a dialog box that warns the user about an error. Normally, an error is considered to be more serious than a warning.
QUESTION_MESSAGE	❓	For a dialog box that asks the user a question. Normally, the question-mark icon is used with a confirm dialog box or an input dialog box. But it is legal to use it with a message dialog box as well.
PLAIN_MESSAGE	no icon	For a plain-looking dialog box. The dialog box contains a message, but no icon.

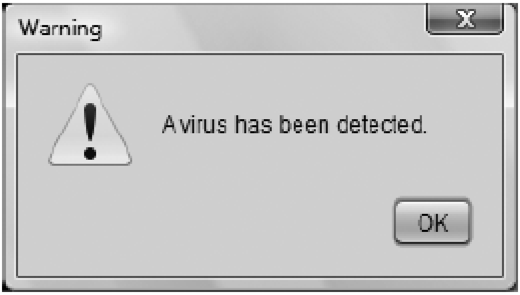
Figure 17.11 Icon options within a JOptionPane dialog

Here's how to call the four-parameter version of showMessageDialog:

JOptionPane.showMessageDialog(<*null-or-container*>, <*message*>, <*title*>, <*icon_constant*>);

Here's an example four-argument showMessageDialog call and the resulting dialog:

```
JOptionPane.showMessageDialog(null, "A virus has been detected.", "Warning",
    JOptionPane.WARNING_MESSAGE);
```



A virus has been detected.

17.14 Distinguishing Between Multiple Events

Now that you understand the basic building blocks of GUI programming (JFrame and JOptionPane windows; JLabel, JTextField, and JButton components), you're prepared to consider more complex situations that GUI programmers encounter. In this section, you'll learn how to use a single listener to distinguish between two different component events.

The getSource Method

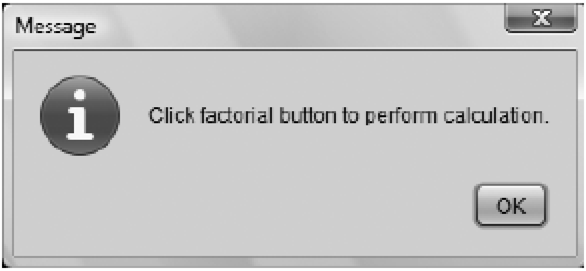
Suppose that you register a listener with two components. When the listener hears an event, you'll probably want to determine which component fired the event. That way, you can customize your event handling: Do one thing if component X fired the event, and do another thing if component Y fired the event.

From within a listener, how can you determine the source of an event? In other words, how can you identify the component that fired an event? Call getSource, of course! More specifically, within the actionPerformed method, use the actionPerformed method's ActionEvent parameter to call getSource. The getSource method returns a reference to the component whose event was fired. To see which component that was, use = = to compare the returned value with the components in question. For example, in the below code fragment, we compare the returned value to a button component named okButton.

```
   public void
actionPerformed(ActionEvent e)
   {
   if (e.getSource() = = okButton)
   {
   ...
```

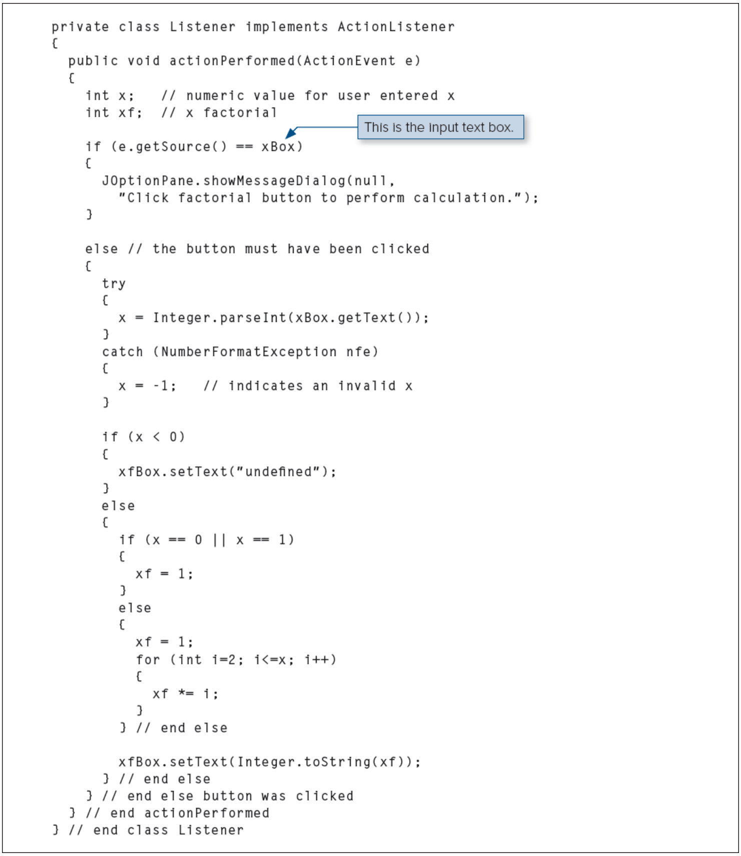
Improved FactorialButton Program

Remember the FactorialButton program from Figure 17.8? It calculated the factorial of a user-entered number. The calculations were triggered by the user clicking the factorial button or the user pressing Enter in the input text box. With our simple first-cut FactorialButton program, we didn't bother to distinguish between the button-click event and the text-box-enter event. Let's now improve the program by having the different events trigger different results. The button click will still display the factorial, but the text box enter will display this dialog-box message:



Click factorial button to perform calculation.

See Figure 17.12. It shows the Listener class for our new and improved FactorialButton program. We're showing only the Listener class because the rest of the program hasn't changed. If you want to see the rest of the program, refer back to Figure 17.8. In our new Listener class, note how we call getSource and compare its returned value to xBox. xBox is the text box component that holds the user’s entry for x. If getSource returns xBox, we call showMessageDialog and display the above dialog message.



```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    int x;   // numeric value for user entered x
    int xf;  // x factorial

    if (e.getSource() == xBox)
    {
      JOptionPane.showMessageDialog(null,
        "Click factorial button to perform calculation.");
    }

    else // the button must have been clicked
    {
      try
      {
        x = Integer.parseInt(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;    // indicates an invalid x
      }

      if (x < 0)
      {
        xfBox.setText("undefined");
      }
      else
      {
        if (x == 0 || x == 1)
        {
          xf = 1;
        }
        else
        {
          xf = 1;
          for (int i=2; i<=x; i++)
          {
            xf *= i;
          }
        } // end else

        xfBox.setText(Integer.toString(xf));
      } // end else
    } // end else button was clicked
  } // end actionPerformed
} // end class Listener
```

This is the input text box.

Figure 17.12 Modified Listener class for the FactorialButton program

17.15 Using

getActionCommand to Distinguish Between Multiple Events

In this section, we continue our discussion of distinguishing between multiple events. But instead of calling getSource, this time we call getActionCommand.

getSource Is Somewhat Limited

In Figure 17.12's Listener class, we call getSource to identify the component whose event was fired. That works fine most of the time, but not always. Note the following cases where calling getSource is inadequate:

1. If the event-firing components are in a different class from the listener class.

   A listener class's getSource method can retrieve the component responsible for the fired event, but there is no way to identify the type of the returned component because that requires comparing the returned component with the original components (using ==). If the original components are in a -different class and private, using them in the listener class generates a compile-time error.

2. If there's a need to have a *modal* component.

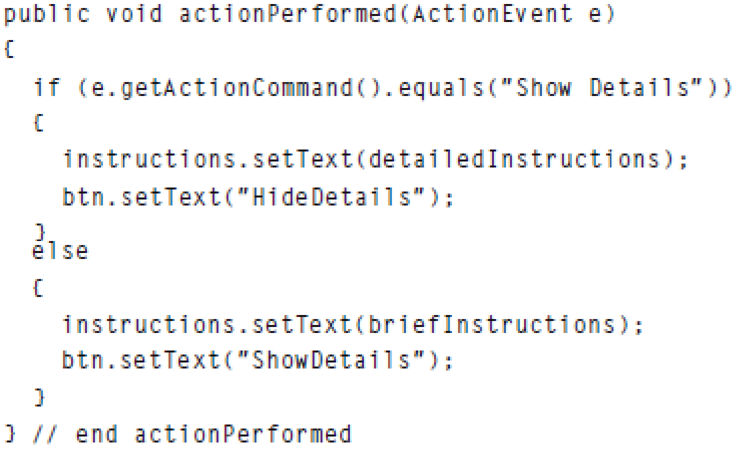
A modal component is a component with more than one state or status. For example, suppose there's a button whose label toggles between "Show Details" and "Hide Details." The two labels correspond to two different modes of operation—in one mode details are shown, and in another mode details are hidden. If a modal button is clicked, getSource can retrieve the button, but it cannot retrieve the button’s mode. In the show details/hide details example, getSource cannot directly determine whether the button’s mode is show details or hide details.

getActionCommand to the Rescue

If you need to identify an event from within a listener and getSource is inadequate, you have another - alternative—the getActionCommand. The getActionCommand method returns the “action command” associated with the component whose event was fired. Typically, the action command is the component’s label. For example, the default action command for a button is the button’s label.

Let’s revisit the case where a button’s label toggles between “Show

Details" and "Hide Details." In the following code fragment, assume that instructions is a label component, detailedInstructions and briefInstructions are string local variables, and btn is the "Show Details/Hide Details" button. Note how getActionCommand determines the button's mode by retrieving the button's label:



```
public void actionPerformed(ActionEvent e)
{
  if (e.getActionCommand().equals("Show Details"))
  {
    instructions.setText(detailedInstructions);
    btn.setText("HideDetails");
  }
  else
  {
    instructions.setText(briefInstructions);
    btn.setText("ShowDetails");
  }
} // end actionPerformed
```

17.16 Color

So far in this chapter, all of our components have been simple in terms of color—black text on white background or black text on light-gray background. It’s time to add some color. You should get used to adding color to most of your GUI applications. After all, color can enhance a user’s experience with a program by providing visual cues and visual appeal. Remember, color is fun!

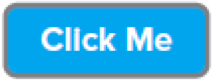
Color Methods

Most GUI components are composed of two colors. The *foreground color* is the color of the text, and the *background color* is the color of the area behind the text. Let’s jump right into an example that shows you how to set the colors. This code fragment creates a dark-green button with white text:

```
JButton btn = new JButton("Click Me");
btn.setBackground(Color.GREEN.darker())
btn.setForeground(Color.WHITE);
```

And here’s what the dark-green button

with white text looks like:



The setBackground and setForeground methods are mutator methods. Here are the API headings and descriptions for their associated accessor methods:

public Color getBackground()

Returns the component’s background color.

public Color getForeground()

Returns the component’s foreground color.

Here’s an example that uses the getBackground and getForeground methods with a text box:

```
JTextField nameBox = new JTextField();
Color originalBackground = nameBox.getBackground();
Color originalForeground = nameBox.getForeground();
```

Why might you want to save a text box's original colors? As a visual cue, you might want to change a text box's colors when the user enters something invalid. And when the user fixes the entry, you'd change back to the original colors. In order to do that, you need to retrieve and save the original colors when the window is first loaded.

You've now seen color examples with a button and a text box. Color works the same way for most other components. An exception is the JLabel component. Its background is transparent by default, so if you apply color to it, you won't see the color. To change a label's background color, you first have to make it opaque by having the label reference variable call setOpaque(true). After that, if you call setBackground(<*color*>), you'll see the specified color.

Color Named Constants

Let’s now talk about color values. You can specify color values with named constants or with instantiated Color objects. We’ll start with named constants.

The Color class defines this set of named constants:

```
Color.BLACK         Color.GREEN          Color.RED
Color.BLUE          Color.LIGHT_GRAY     Color.WHITE
Color.CYAN          Color.MAGENTA        Color.YELLOW
Color.DARK_GRAY     Color.ORANGE
Color.GRAY          Color.PINK
```

These named constants are class members, so we access them using <*class name*> dot syntax, where the class name is Color. For darker or brighter shades, chain darker() or brighter() method calls.

The Color class is in the java.awt package, so don’t forget to import that package when working with colors.

Color Objects

To obtain a color that is not in the Color class’s list of named constant colors, instantiate a Color object with a specified mixture of red, green, and blue. Here's the Color constructor call syntax:

new Color(<*red 0–255*>, <*green 0–255*>, <*blue 0–255*>)

Each of the three Color constructor arguments is an int value between 0 and 255. The int value represents an amount of color, with 0 indicating no color and 255 indicating the maximum amount of color. For example, this line sets a button's background color to a dark magenta:

button.setBackground(new Color(128, 0, 128));

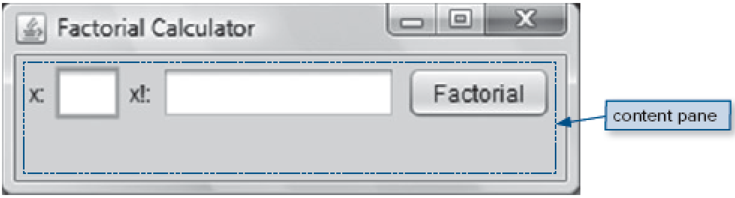
The instantiated Color object uses half the maximum for red (128), no green (0), and half the maximum for blue (128). For the brightest magenta, increase the red and blue values from 128 to 255.

White light is the combination of all colors,[4] so new Color(255, 255, 255) produces white. Black is the absence of all colors, so new Color(0, 0, 0) produces black.

This technique of creating a color by mixing specified amounts of red, green, and blue is used by many programming languages. The red, green, blue 3-tuple is commonly referred to as an *RGB value*. When coming up with RGB values for your programs, it's perfectly acceptable to use trial and error, but to save time, you may want to visit an RGB color table online. For example—http://web.njit.edu/~kevin/rgb.txt.html.

JFrame Background Color

Setting the background color for a JFrame window is slightly trickier than setting it for a component. First, you have to get the JFrame's *content pane,* and then you have to apply the background color to it. As shown below, the content pane is the inner part of the JFrame.



While the JFrame class handles perimeter features such as window dimensions, the title bar, and the close-out button, the content pane handles interior features such as components, layout, and background color. So when you add components, set the layout, and set the background color, you do it to the content pane, not the JFrame. These three statements illustrate what we're talking about:

```
    getContentPane().add(btn);
    getContentPane().setLayout(new
FlowLayout());
    getContentPane().setBackground(Color.YE
```

In versions of Java prior to Java 5.0, JFrame's getContentPane method was required for all three tasks—adding a component, setting the layout, and setting the window's background color. With the advent of Java 5.0, the Java language designers made things easier for the first two tasks. Now, if you want to add a component or set the layout, as an option, you may omit the call to getContentPane. In other words, this works:

```
add(btn);
setLayout(new FlowLayout());
```

The reason this code works is that with the current version of Java, JFrame's add and setLayout methods automatically get the content pane behind the scenes. And the retrieved content pane is used for the ensuing add and setLayout operations. So which is better—getContentPane().add(btn) or just add(btn)? They are functionally equivalent, but the second one is generally preferred because it's less cluttered. Ditto for the setLayout method call.

For setting the window's background color, the current version of Java still requires that you call getContentPane before calling setBackground. If you call setBackground without calling ⚠ getContentPane, it sets the JFrame's background color, not the content pane's background color. And because the content pane sits on top of the JFrame, the JFrame's color is covered up and not seen.

Now you know that setting a window's background color requires getContentPane. Similarly, getting a window's background color requires getContentPane. For example:

```
Color saveColor = getContentPane().getBackground();
```

ColorChooser Program

Let's put what you've learned about

color into practice by using it within a complete program. In our ColorChooser program, we implement light gray and green buttons that set the window's background color to gray or dark green, respectively. See Figure 17.13 to get an idea of what we're talking about.

See the ColorChooser program listing in Figures 17.14a, 17.14b, and 17.14c. Most of the code should already make sense because its structure mirrors the structure in our previous GUI programs. We'll focus on the new code—the color code.

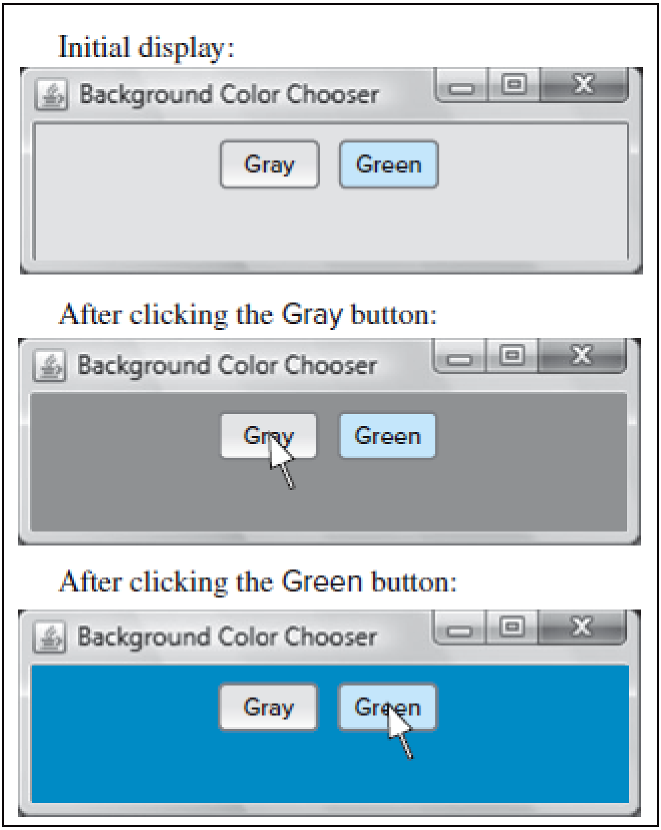
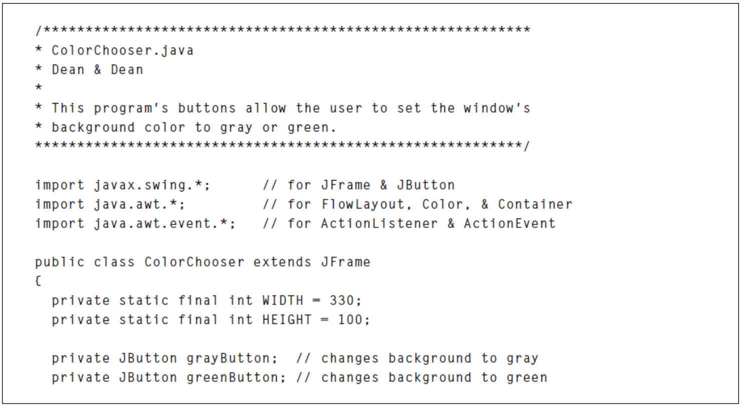

Initial display:
Background Color Chooser
Gray
Green
After clicking the Gray button:
Background Color Chooser
Gray
Green
After clicking the Green button:
Background Color Chooser
Gray
Green

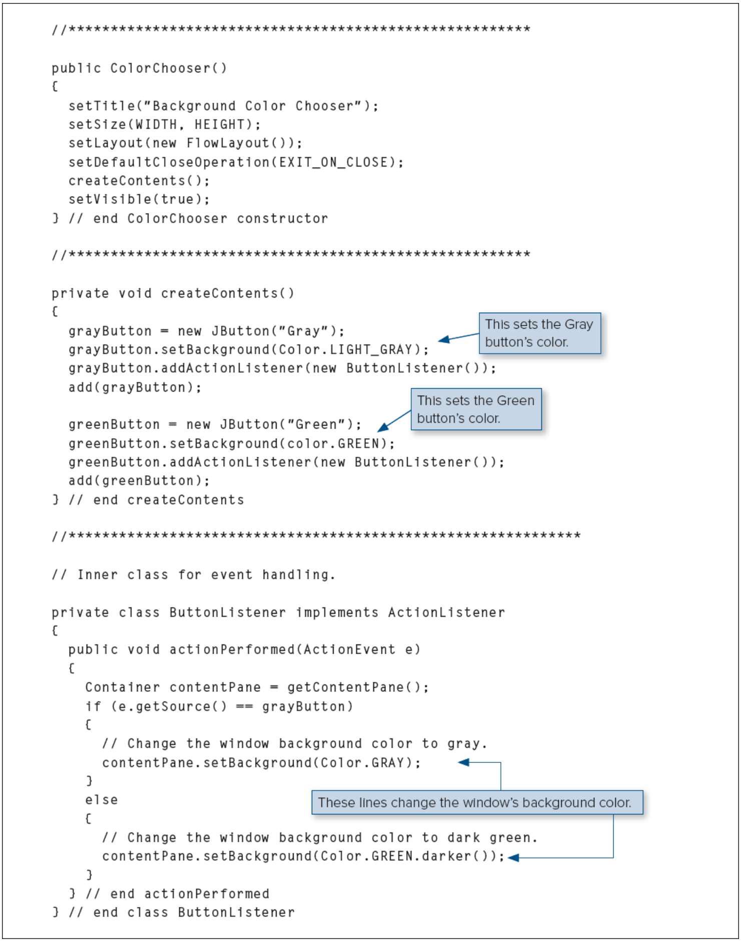
Figure 17.13 Sample session for the ColorChooser program



```
/*************************************************************
* ColorChooser.java
* Dean & Dean
*
* This program's buttons allow the user to set the window's
* background color to gray or green.
*************************************************************/

import javax.swing.*;       // for JFrame & JButton
import java.awt.*;          // for FlowLayout, Color, & Container
import java.awt.event.*;    // for ActionListener & ActionEvent

public class ColorChooser extends JFrame
{
  private static final int WIDTH = 330;
  private static final int HEIGHT = 100;

  private JButton grayButton;  // changes background to gray
  private JButton greenButton; // changes background to green
```

Figure 17.14a ColorChooser program—

part A



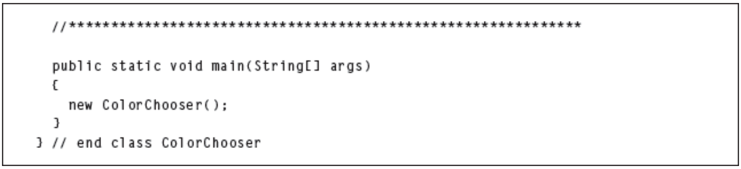
```
//**********************************************************

public ColorChooser()
{
  setTitle(”Background Color Chooser”);
  setSize(WIDTH, HEIGHT);
  setLayout(new FlowLayout());
  setDefaultCloseOperation(EXIT_ON_CLOSE);
  createContents();
  setVisible(true);
} // end ColorChooser constructor

//**********************************************************

private void createContents()
{
  grayButton = new JButton(”Gray”);
  grayButton.setBackground(Color.LIGHT_GRAY);
  grayButton.addActionListener(new ButtonListener());
  add(grayButton);

  greenButton = new JButton(”Green”);
  greenButton.setBackground(color.GREEN);
  greenButton.addActionListener(new ButtonListener());
  add(greenButton);
} // end createContents

//****************************************************************

// Inner class for event handling.

private class ButtonListener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    Container contentPane = getContentPane();
    if (e.getSource() == grayButton)
    {
      // Change the window background color to gray.
      contentPane.setBackground(Color.GRAY);
    }
    else
    {
      // Change the window background color to dark green.
      contentPane.setBackground(Color.GREEN.darker());
    }
  } // end actionPerformed
} // end class ButtonListener
```

This sets the Gray button’s color.

This sets the Green button’s color.

These lines change the window’s background color.

Figure 17.14b ColorChooser program—part B



```
  //****************************************************************

  public static void main(String[] args)
  {
    new ColorChooser();
  }
} // end class ColorChooser
```

Figure 17.14c ColorChooser program—part C

Note the difference between the setBackground calls in the createContents method and the setBackground calls in the actionPerformed method. In createContents, we’re dealing with the gray and green button components, so it is not necessary to call getContentPane prior to calling setBackground. In actionPerformed, we’re dealing with the JFrame window, so it is necessary to call getContentPane prior to calling setBackground.

Finally, note that instead of using a color constant, in the createContents method we could generate the green for the button’s background by creating a new color object, like this:

```
greenButton.setBackground(new Color(0, 255, 0));
```

And in the actionPerformed method we could generate the darker green color for the window's background by creating a new Color object, like this:

```
   ContentPane.setBackground(new
Color(0, 192, 0));
```

17.17 How GUI Classes Are Grouped Together

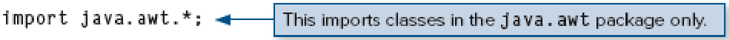
Throughout this chapter, you've used Java's prebuilt GUI classes from Oracle's Java API library. For example, you used the JFrame class for creating a window, the JButton class for creating a button, and the Color class for creating a color. In this section, we describe how Java API prebuilt GUI classes are grouped and organized.

Subpackages

The Java API is a huge class library that adds functionality to the core Java language. To simplify things, the classes are organized into a hierarchy of packages where each package contains a group of classes. To avoid having too many classes in one package, packages are often split into *subpackages*. A subpackage is a group of classes from within a larger group of classes. For example, rather than putting all the GUI classes (and there are lots of them!) within the java.awt package, Java language designers split off the GUI event-handling classes and put them in their own subpackage, java.awt.event. To import all the classes in the java.awt package and the java.awt.event subpackage, do this:

```
import java.awt.*;
import java.awt.event.*;
```

Because the java.awt.event subpackage contains java.awt in its name, is it OK to omit the java.awt.event import statement and do just the following?



```
import java.awt.*;
```

This imports classes in the java.awt package only.

No—you must import java.awt and java.awt.event separately. Think of the java.awt package and the java.awt.event subpackage as completely separate entities. The fact that they share the common ⚠name “java.awt” is irrelevant as far as the compiler is concerned. The compiler treats them as separate packages. So then why the shared name? The shared name helps programmers remember that the classes in java.awt.event are conceptually related to the classes in java.awt.

We’ve been referring to java.awt.event as a *“subpackage”*. It’s just as common to refer to it as a *“package”*. We’ll use both terms because both are valid.

The AWT and Swing Libraries

In Sun’s first Java compiler, all GUI classes were bundled into one library known as the Abstract Windowing

Toolkit (AWT). The AWT's GUI commands generate GUI components that look different on different platforms. In other words, if your program instantiates an AWT button component, the button will have a Macintosh look and feel if the program is run on a Macintosh computer, but a Windows look and feel if the program is run on a Windows computer.[5] That leads to portability issues. Your programs are still portable in the sense that they'll run on different platforms. But they'll run differently on different platforms. If you have a persnickety customer who demands one precise appearance on all platforms, then AWT components probably won't be satisfactory.

One of Java's strongest selling points was (and is) its portability, so soon after Java's initial release, the Java language designers proceeded to develop a set of more portable GUI

components. They put their new, more-portable components in a brand new library named Swing. To make the relationship clear between the new Swing components and the AWT components, they used the same component names except that they prefaced the new Swing components with a “J.” For example, the AWT has a Button component, so Swing has a JButton component.

The AWT GUI components are known as *heavyweight components,* while the Swing GUI components are known as *lightweight components*. The AWT components are said to be heavyweight because they are built by the native environment that runs the bytecode. (The term native environment refers to the low-level instructions inherent to a particular computer platform, such as Windows or Macintosh.) For each AWT component, the bytecode asks the native environment to create the component, and it's up to the native environment to find native code for the requested component. That's the reason AWT components have a different look and feel when run on different computer platforms. On the other hand, Swing components are said to be lightweight because they're built with Java code. Being built with Java code means they're "light" enough to move to different platforms and have the same look and feel everywhere.

The Swing library includes more than just GUI component classes. It adds lots of functionality to the AWT, but it does not replace the AWT entirely. Today, Java GUI application programmers use both libraries—the AWT and Swing. The primary AWT packages are java.awt and java.awt.event. The primary Swing package is javax.swing. The "x" in javax stands for "extension" because the javax packages (javax.swing is one of several javax packages) are considered to be a major extension to the core Java platform.

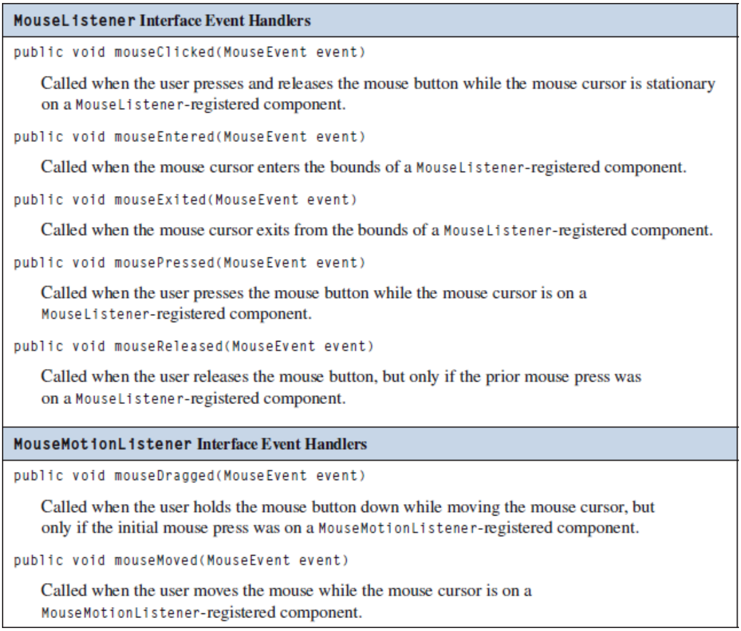
17.18 Mouse Listeners and Images (Optional)

The Java API provides several different types of listeners. Earlier in this chapter, you learned about the most common listener—the ActionListener. You should use the ActionListener for events where the user does something to a component, such as clicking a button or pressing Enter within a text box. In this section, you'll learn about mouse listeners. As the name implies, you should use mouse listeners for events where the user does something with the mouse. Also in this section, you'll learn about images (pictures). You'll learn how to display an image and drag an image with your mouse.

Mouse Listeners

In creating a mouse listener, you use the same basic steps that you use for the ActionListener—you define a listener class, you define an event handler method(s) within the listener class, and you register your listener class with a component. Although the same basic steps are used, mouse listeners are slightly more complicated than the ActionListener. There are several different types of mouse listeners, and each type of mouse listener handles multiple types of mouse events.

We describe two mouse listener types, and they are defined by their two interfaces—MouseListener and MouseMotionListener. Figure 17.15 shows the API headings and descriptions for the methods defined by the two interfaces. Read through the API headings and descriptions to get an idea of what's possible in terms of mouse event handling.



`MouseListener` Interface Event Handlers	
`public void mouseClicked(MouseEvent event)`	Called when the user presses and releases the mouse button while the mouse cursor is stationary on a `MouseListener`-registered component.
`public void mouseEntered(MouseEvent event)`	Called when the mouse cursor enters the bounds of a `MouseListener`-registered component.
`public void mouseExited(MouseEvent event)`	Called when the mouse cursor exits from the bounds of a `MouseListener`-registered component.
`public void mousePressed(MouseEvent event)`	Called when the user presses the mouse button while the mouse cursor is on a `MouseListener`-registered component.
`public void mouseReleased(MouseEvent event)`	Called when the user releases the mouse button, but only if the prior mouse press was on a `MouseListener`-registered component.
`MouseMotionListener` Interface Event Handlers	
`public void mouseDragged(MouseEvent event)`	Called when the user holds the mouse button down while moving the mouse cursor, but only if the initial mouse press was on a `MouseMotionListener`-registered component.
`public void mouseMoved(MouseEvent event)`	Called when the user moves the mouse while the mouse cursor is on a `MouseMotionListener`-registered component.

Figure 17.15 API headings and descriptions for the methods in the MouseListener and MouseMotionListener interfaces

As a programmer, you don't have to worry about calling the mouse event handler methods. They're called automatically when their associated mouse events occur. For example, if the user presses the mouse button while the mouse cursor is on a MouseListener-registered component, the JVM automatically calls the mousePressed event handler.

In the upcoming program, the goal is to enable a user to drag an image across a window using the mouse. To do that, you need to detect the mouse being pressed and moved (i.e., dragged) while the mouse cursor is on the image. And to do that, you need to register a mouse listener. But you can register a mouse listener only with a component, not with an image. So what's the solution? You're already familiar with some components—JLabel, JTextField, and JButton. Those classes are component classes because they are descendants of the Component class. There's another component class that's a bit different. It doesn't feel like a component in the normal sense of the word, but it's a Java component nonetheless (because it's a descendant of the Component class), and it works great for handling mouse events. So what is the mystery component? JPanel!

Think of a JPanel object as a generic storage area for other objects. More formally, the JPanel class is a descendant of the Container class, and as such, it's a container and you can add objects to it. In the next chapter, you'll add Swing components (JLabel, JTextField, and so on) to JPanel - containers. In the upcoming program example, you add an image object to a JPanel container. By surrounding the image with a JPanel container, you provide a platform that mouse listeners can attach to. In the upcoming program example, the JPanel listeners allow you to detect mouse events on the image object.

The DragSmiley Program

See Figure 17.16. It contains a driver

class and a sample session for a DragSmiley program. As indicated in the sample session, the program initially displays a smiley face in the top-left corner of the program's window. If the user presses the mouse button, the smiley image changes to a scared image (presumably because the smiley is apprehensive of what the user might do to it). When the user releases the mouse button, the scared image changes back to the smiley image. If the mouse cursor resides on the image and the user drags the mouse, the image follows the mouse cursor.

Study Figure 17.16's DragSmiley constructor. In it, the following two statements instantiate a JPanel container named smileyPanel and add the JPanel container to DragSmiley's window.

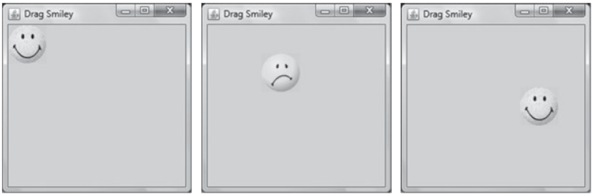
```
smileyPanel = new SmileyPanel();
add(smileyPanel);
```

See the SmileyPanel class in Figures 17.17a, 17.17b, and 17.17c. The SmileyPanel class is where the bulk of the program's logic is. We'll describe the SmileyPanel class by first focusing on the listeners. Note how the SmileyPanel constructor creates the mouse listeners and adds them to the JPanel container. Note the mouse listener class headings, repeated here for your convenience:

```
private class ClickListener extends MouseAdapter
private class DragListener extends MouseMotionAdapter
```

The extends clauses indicate inheritance from the MouseAdapter and MouseMotionAdapter classes. For each event handling interface with more than one method, the Java API provides an associated class that already implements the interface's methods for you. Those classes are called *adapter classes*. The MouseAdapter class implements the MouseListener interface’s methods. -Likewise, the MouseMotionAdapter class implements the MouseMotionListener interface’s methods. Adapter classes don’t do much. They simply implement their associated interface’s methods as dummy methods, like this:

```
/*********************************************************
* DragSmiley.java
* Dean & Dean
*
* This program displays a smiley face image.
* When the user presses the mouse, the image changes to a
* scared image. The user can drag the image.
********************************************************/

import javax.swing.JFrame;

public class DragSmiley extends JFrame
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 250;
  private SmileyPanel smileyPanel;      // drawing panel

  //*****************************************************

  public DragSmiley()
  {
    setTitle(”Drag Smiley”);
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    smileyPanel = new SmileyPanel();
    add(smileyPanel);
    setVisible(true);
  } // end DragSmiley constructor

  //**************************************

  public static void main(String[] args)
  {
    new DragSmiley();
  }
} // end class DragSmiley
```

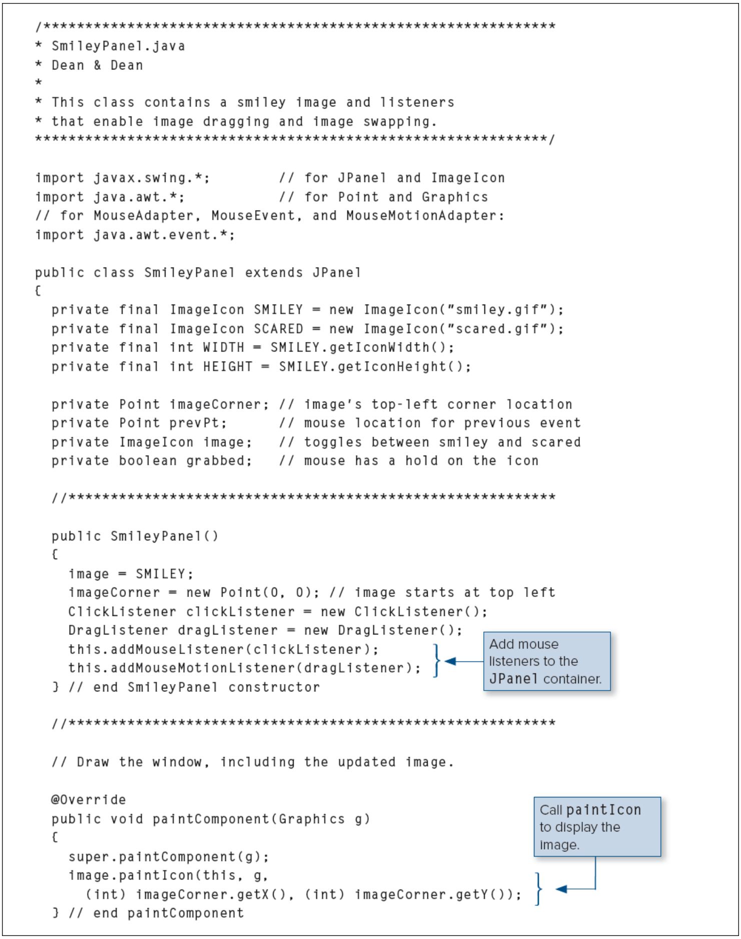


Initial display:

While dragging smiley:

After releasing mouse button:

Figure 17.16 Driver class and sample output for the DragSmiley program

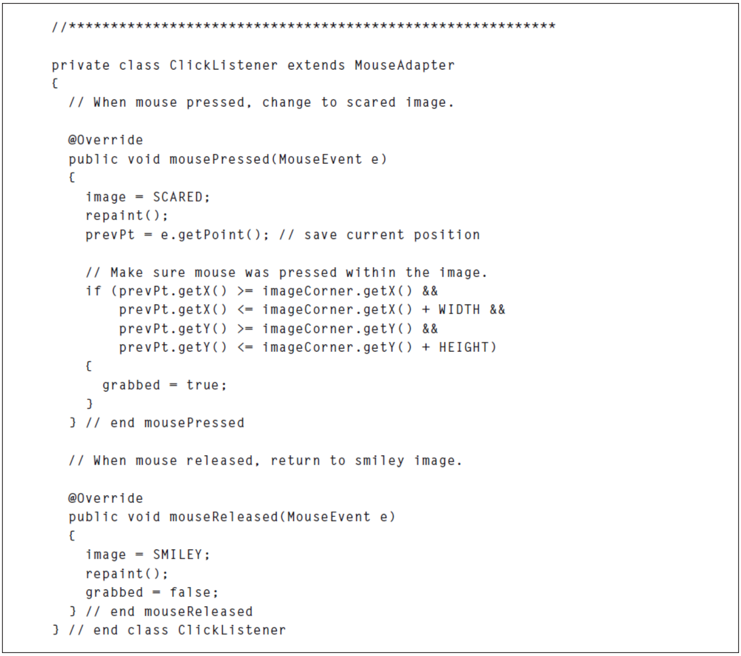


```
/*************************************************************
* SmileyPanel.java
* Dean & Dean
*
* This class contains a smiley image and listeners
* that enable image dragging and image swapping.
*************************************************************/

import javax.swing.*;        // for JPanel and ImageIcon
import java.awt.*;           // for Point and Graphics
// for MouseAdapter, MouseEvent, and MouseMotionAdapter:
import java.awt.event.*;

public class SmileyPanel extends JPanel
{
  private final ImageIcon SMILEY = new ImageIcon("smiley.gif");
  private final ImageIcon SCARED = new ImageIcon("scared.gif");
  private final int WIDTH = SMILEY.getIconWidth();
  private final int HEIGHT = SMILEY.getIconHeight();

  private Point imageCorner; // image's top-left corner location
  private Point prevPt;      // mouse location for previous event
  private ImageIcon image;   // toggles between smiley and scared
  private boolean grabbed;   // mouse has a hold on the icon

  //**********************************************************

  public SmileyPanel()
  {
    image = SMILEY;
    imageCorner = new Point(0, 0); // image starts at top left
    ClickListener clickListener = new ClickListener();
    DragListener dragListener = new DragListener();
    this.addMouseListener(clickListener);
    this.addMouseMotionListener(dragListener);
  } // end SmileyPanel constructor

  //**********************************************************

  // Draw the window, including the updated image.

  @Override
  public void paintComponent(Graphics g)
  {
    super.paintComponent(g);
    image.paintIcon(this, g,
      (int) imageCorner.getX(), (int) imageCorner.getY());
  } // end paintComponent
```

Add mouse listeners to the `JPanel` container.

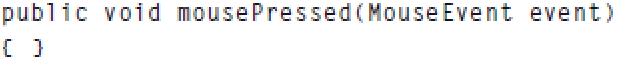
Call `paintIcon` to display the image.

Figure 17.17a DragSmiley program's SmileyPanel class—part A



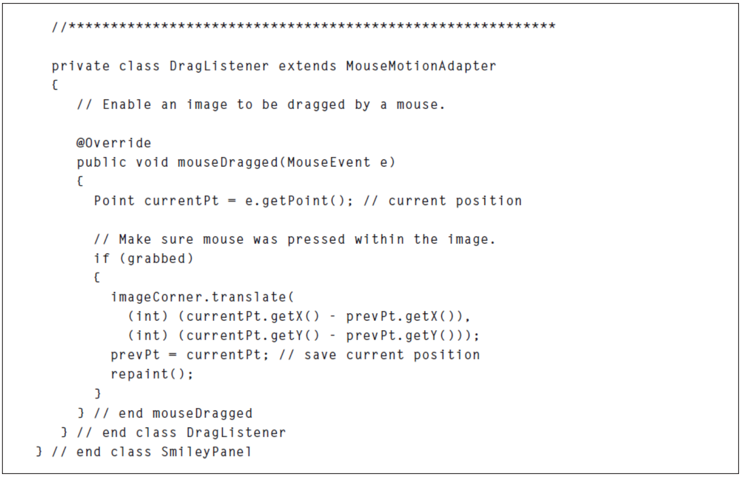
```
//*************************************************************

private class ClickListener extends MouseAdapter
{
  // When mouse pressed, change to scared image.

  @Override
  public void mousePressed(MouseEvent e)
  {
    image = SCARED;
    repaint();
    prevPt = e.getPoint(); // save current position

    // Make sure mouse was pressed within the image.
    if (prevPt.getX() >= imageCorner.getX() &&
        prevPt.getX() <= imageCorner.getX() + WIDTH &&
        prevPt.getY() >= imageCorner.getY() &&
        prevPt.getY() <= imageCorner.getY() + HEIGHT)
    {
      grabbed = true;
    }
  } // end mousePressed

  // When mouse released, return to smiley image.

  @Override
  public void mouseReleased(MouseEvent e)
  {
    image = SMILEY;
    repaint();
    grabbed = false;
  } // end mouseReleased
} // end class ClickListener
```

Figure 17.17b DragSmiley program's SmileyPanel class—part B



```
public void mousePressed(MouseEvent event)
{ }
```

To implement a listener that detects the mouse being pressed, you extend the MouseAdapter class and provide an overriding mousePressed method. For an example, see Figure 17.17b. As an alternative, you can implement a listener using an interface rather than an adapter. But remember that an interface is a contract, and when you implement an interface, you're required to provide methods for all the interface's methods. So if you wanted to replace the SmileyPanel class's adapters with interfaces, you'd have to provide dummy methods for methods you won't use. On the other hand, when you extend adapter classes, you just override those methods you will use.



```
  //************************************************************

  private class DragListener extends MouseMotionAdapter
  {
     // Enable an image to be dragged by a mouse.

     @Override
     public void mouseDragged(MouseEvent e)
     {
       Point currentPt = e.getPoint(); // current position

       // Make sure mouse was pressed within the image.
       if (grabbed)
       {
         imageCorner.translate(
           (int) (currentPt.getX() - prevPt.getX()),
           (int) (currentPt.getY() - prevPt.getY()));
         prevPt = currentPt; // save current position
         repaint();
       }
     } // end mouseDragged
   } // end class DragListener
} // end class SmileyPanel
```

Figure 17.17c DragSmiley program's SmileyPanel class—part C

Displaying an Image

It's now time to see how the

SmileyPanel class draws its images. At the top of the class, the SMILEY and SCARED named constants are initialized as follows:

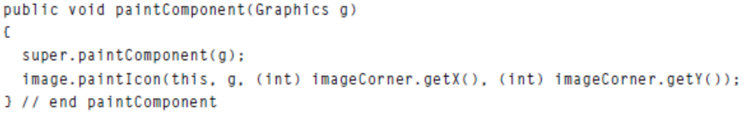
```
  final private ImageIcon SMILEY =
new ImageIcon("smiley.gif");
  final private ImageIcon SCARED =
new ImageIcon("scared.gif");
```

The ImageIcon constructor creates an image object from its passed-in filename parameter. So in the above code fragment, two image objects are created from the smiley.gif and scared.gif files, respectively.[6]

In the SmileyPanel constructor, the mousePressed event handler, and the mouseReleased event handler, note how SMILEY and SCARED get assigned into the image instance variable. Those assignments are what cause the image to change when the user presses the mouse button and releases it.

The JPanel class has a

paintComponent method that's in charge of drawing Swing components (e.g., text boxes and buttons) within the JPanel container. But it doesn't handle drawing lines, shapes, or images. To draw those things, you need to provide an overriding paintComponent method with calls to graphics methods. For example, here is SmileyPanel's overriding paintComponent method:



```
public void paintComponent(Graphics g)
{
  super.paintComponent(g);
  image.paintIcon(this, g, (int) imageCorner.getX(), (int) imageCorner.getY());
} // end paintComponent
```

Note the paintComponent method's g parameter. It's a Graphics object, and it's used to call graphics methods within the paintComponent method. For example, the image.paintIcon method call draws image (a smiley face or a scared face), and it requires a Graphics object, g, for its second argument. In calling the paintIcon method, you provide three arguments in addition to the Graphics argument:

(1) an image observer, which listens for the completion of the image being loaded, (2) the x coordinate of the image's top-left corner, and (3) the y coordinate of the image's top-left corner.

The first argument is an image observer component that manages repainting as the image file downloads initially or as an animated image file changes. It's common for an image to take longer to display than the rest of the window because image files tend to be relatively large. If you want to do something special when the image finishes loading, then add an overriding imageUpdate event handler method to the image observer's class. In the DragSmiley program, we don't implement an imageUpdate event handler. In the above code fragment, we use this for the image observer component, where this refers to the -SmileyPanel calling object.

SmileyPanel is a subclass of JPanel, and JPanel implements the automatic repaint functionality required by an image observer component.

In the above code fragment, note the super.paintComponent(g) method call. You should always include that call as the first statement within an overriding paintComponent method. Without it, the background for paintComponent's associated object might be displayed improperly.

Notice that there's no explicit call to the DragSmiley program's paintComponent method. You should never call the paintComponent method directly. Instead, you should call the repaint method and let the repaint method call the paintComponent method for you. The repaint method waits until the program's window is properly prepared to handle the paintComponent method. Note in the SmileyPanel class how repaint is called at the bottom of the three event handlers. That's where there's a need to redraw the image. By the way, in addition to calling paintComponent whenever repaint is called, the JVM calls paintComponent automatically when the program starts up and whenever a user does something to alter the program's window (e.g., when the user resizes the window or moves another window off the window).

Summary

- The JFrame class should be used as the superclass for most of your GUI application windows.
- The JFrame class implements all the standard window features such as a border, a title bar, a minimize button, a close-window button (the "X"), the ability to resize the window, and so on.

- JLabel is a read-only component; the user simply reads the label's message.
- The JTextField component allows the user to enter text into a text box.
- When the user interacts with a component (e.g., when the user clicks a button or presses enter while in a text box), the component fires an event.
- If a component has a listener attached to it, the fired event is "heard" by the listener and consequently handled by the listener.
- A listener handles an event by executing its actionPerformed event-handler method.
- Listeners often are implemented with the ActionListener interface. An interface is a class-like entity

whose methods are all empty. If a programmer uses an interface to derive a new class, the compiler requires the new class to implement methods for all the interface’s methods.

- If a class is limited in its scope such that it is needed by only one other class, then you should define the class as an inner class (a class inside another class).
- An anonymous inner class is an inner class without a name.
- To display a simple window with a message, call JOptionPane’s showMessageDialog method.
- To identify the component whose event was fired, use the actionPerformed method’s ActionEvent parameter to call getSource or getActionCommand.
- To adjust a GUI component’s text

color, call setForeground. To adjust the color behind the text, call setBackground.

- To adjust a window's background color, call the content pane's setBackground method.
- To detect and handle mouse events, use the MouseAdapter and MouseMotionAdapter classes, which implement the MouseListener and MouseMotionListener interfaces, respectively.

Review Questions

---

§17.2 Event-Driven Programming Basics

1. What is a listener?
2. What is an event handler?

§17.3 A Simple Window Program

3. Write a statement that adds functionality to a program's close-window button such that when the close-window button is clicked, it causes the program to terminate.

§17.4 JFrame Class

4. What is the name of the superclass for classes that contain components?

§17.5 Java Components

5. What package are JButton and many other J-prefixed components defined in?

§17.6 JLabel Component

6. Provide an initialization statement that declares a JLabel reference variable named hello and assigns "Hello World" to the reference variable.

§17.7 JTextField Component

7. Provide an initialization statement that instantiates a 10-character-wide text box object. As part of the initialization, assign the text box object to a reference variable named input.

§17.9 Component Listeners

8. Write a statement that registers a listener reference variable named responder with a component named component.

9. If you want a class to handle an event, what clause must be added to the right side of the class's heading?

10. What is the heading of the one method specified by the ActionListener interface?

§17.10 Inner Classes

11. If a class is limited in scope

such that it is only needed internally within another class, you should define the class to be an __________.

§17.11 Anonymous Inner Classes

12. If you want to implement an event handler with an anonymous inner class, what argument do you give to the addActionListener method to register the listener?

§17.12 JButton Component

13. In the createContents method of the FactorialButton program in Figure 17.8a, what type of object calls the add methods?

14. In the FactorialButton program in Figures 17.8a and 17.8b, what component fires the event that the listener handles?

§17.13 Dialog Boxes and the JOptionPane Class

15. What package contains the JOptionPane class?

16. Write a statement that displays a dialog in the center of the screen. The dialog should display "This is only a test." in the message area, "TEST" in the title area, and no icon.

§17.14 Distinguishing Between Multiple Events

17. Suppose that you have several components registered with the same listener, and the components and listener are defined within the same class. Within the listener, what ActionEvent method should you call to determine which component fires an event?

§17.15 Using getActionCommand to Distinguish Between Multiple Events

18. Assume there's a listener that's been registered for several different buttons. Assume the listener uses an actionPerformed method with an ActionEvent parameter named action. Assume the user clicks one of the registered buttons. Provide a statement that retrieves the text label from the clicked button and assigns the retrieved label to a String variable named buttonLabel.

§17.16 Color

19. Write a statement that sets the text color to blue for a JButton object named button1.

20. How do you get a reference to the container that surrounds all of the components in a JFrame object?

§17.17 How GUI Classes Are

Grouped Together

21. If your program needs the java.awt.event subpackage, you can implicitly import it by importing the java.awt package. (T / F)

Exercises

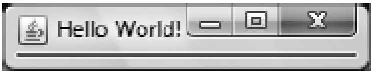
---

1. [after §17.2] Give three examples of how a user might cause an event to be fired.

2. [after §17.3] For each of the following, what Java API package must you import?

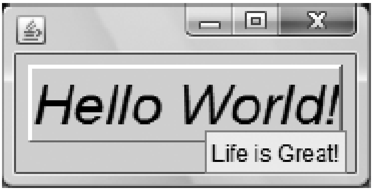
- **a)** JFrame and JLabel
- **b)** FlowLayout

3. [after §17.4] For our previous GUI programs, we've done setup work (setting the title, adding components, and so on) within a constructor. That's generally preferred, but it's not a compiler requirement. For practice purposes, write a minimal, but fully functional, program that displays this:

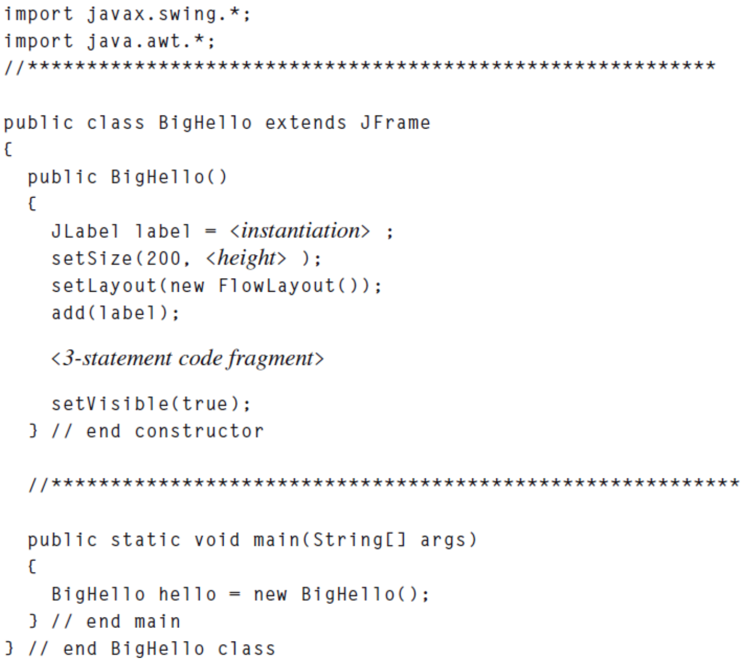


Your program should not include a constructor. It should include only one method—a main method with only five statements (or four statements if you find a shortcut for setting the frame's title).

4. [after §17.6] Provide a complete program that displays this Hello World message:



Note these label characteristics: (1) a raised bevel border, (2) italics, (3) a large font size (30 points), (4) a tool tip that says "Life is Great!" Use this program skeleton as a starting point:



```
import javax.swing.*;
import java.awt.*;
//*******************************************************

public class BigHello extends JFrame
{
  public BigHello()
  {
    JLabel label = <instantiation> ;
    setSize(200, <height> );
    setLayout(new FlowLayout());
    add(label);

    <3-statement code fragment>

    setVisible(true);
  } // end constructor

  //*******************************************************

  public static void main(String[] args)
  {
    BigHello hello = new BigHello();
  } // end main
} // end BigHello class
```

To figure out how to do this, in Java's API, look up the setFont, setBorder, and setToolTipText methods that JLabel inherits from JComponent. For the setFont argument, use Component's getFont to get the default font, and then alter it by using Font's two-parameter deriveFont method in which the first parameter specifies an italic font style and the second parameter specifies a 30-point size. Use JComponent's setBorder method, and for its Border argument, use the appropriate class method from the BorderFactory class.

5. [after §17.7] The width parameter in the JTextField constructor specifies the width of the text box in pixels. (T / F)

6. [after §17.7] What can you do to prevent users from updating a JTextField component?

7. [after §17.9] Write the heading for the method you must define in a class that implements an ActionListener.

8. [after §17.9] The ActionListener interface and the ActionEvent class are in what Java API package?

9. [after §17.9] An interface is a class-like thing whose methods are all empty. If an interface is applied to a class, then the interface acts like a template that the class must conform to. (T / F)

10. [after §17.10] An inner class can access its enclosing class's instance variables directly. (T / F)

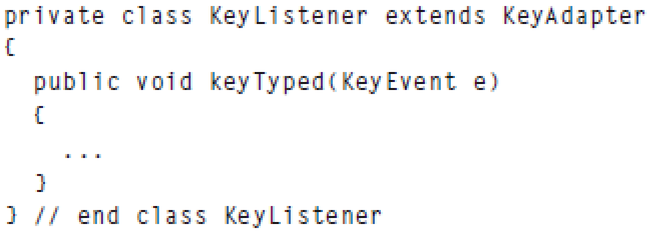
11. [after §17.12] It's appropriate to use an anonymous inner class if you are going to use the class only once. In the Factorial program in Figures 17.8a and 17.8b, we use the listener object twice, so that listener object needed to have a name. However, we used that object's class only once, to instantiate that one object. Therefore, that object's class did not need to have a name, and we could have used an anonymous class to create our listener object. For this exercise, modify the Factorial program to use an anonymous ActionListener class instead of the named Listener class. [Hint: The program is already set up to facilitate this change—it’s mostly cut-and-paste.]

12. [after §17.13] Do you have to create a JFrame window to use a JOptionPane dialog box?

13. [after §17.13] To answer this question, you may need to look up JOptionPane’s showInputDialog and showConfirmDialog methods on Oracle’s Java API website. What does this program do?

```
import javax.swing.JOptionPane;
public class UncertainHello
{
  public static void main(String[] args)
  {
    String name;
    int response;
    do
    {
      name = JOptionPane.showInputDialog(”What’s your name? ”);
      response = JOptionPane.showConfirmDialog(null, ”Are you sure?”);
      if (response == JOptionPane.NO_OPTION)
      {
        name = ”there”;
        break;
      }
    } while (response == JOptionPane.CANCEL_OPTION);

    System.out.println(”Hello ” + name);
  } // end main
} // end class UncertainHello
```

14. [after §17.14] By calling setEnabled(false), you can disable a button and give it a muted appearance and make its listener unresponsive to clicks on it. Modify Figure 17.12’s program so that the factorial button is initially disabled. Enable it only after the user enters a character in the xBox text box. To enable it, create a *key listener* for the xBox text box, and have the key listener’s keyTyped event handler call setEnabled(true). Use the following key listener code skeleton:



```
private class KeyListener extends KeyAdapter
{
  public void keyTyped(KeyEvent e)
  {
    ...
  }
} // end class KeyListener
```

Note extends KeyAdapter in the above class heading. An *adapter* class implements an interface by -providing an empty-bodied method for each method in the interface. In this case, the

KeyAdapter API class implements the KeyListener API interface.

15. [after §17.16] To set a JFrame's background color, what method should you call before calling setBackground?

16. [after §17.17] What do the letters in "awt" stand for?

Review Question Solutions

---

1. A listener is an object that waits for events to occur.

2. An event handler is a method that responds to an event.

3. setDefaultCloseOperation(EXIT_ON_CLO

4. The superclass for objects that contain other objects is the Container class.

5. Many J-prefixed components

are defined in the javax.swing package.

6. JLabel hello = new JLabel("Hello World!");

7. JTextField input = new JTextField(10);

8. component.addActionListener(responde

9. For a class to handle an event, add this to the right side of the class’s heading:

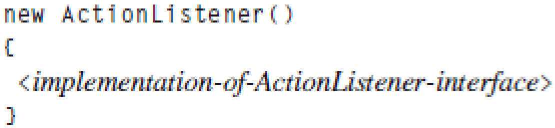
implements ActionListener

10. The heading of the method specified by the ActionListener interface is:

public void actionPerformed(ActionEvent e)

11. If a class is limited in scope such that it is only needed internally within another class, you should define the class to be an inner class.

12. The argument to give to the addActionListener method to register an anonymous listener class is



```
new ActionListener()
{
 <implementation-of-ActionListener-interface>
}
```

13. The object that calls the add methods is a JFrame object.

14. It's ambiguous. It could be either xBox or btn.

15. The package that contains the JOptionPane class is the javax.swing package.

16. This code generates the asked-for dialog box:

> JOptionPane.showMessageDialog(n
>
> "This is only a test.", "TEST", JOptionPane.PLAIN_MESSAGE);

17. To identify the firing component, call the getSource method.

18. buttonLabel = action.getActionCommand();

19. button1.setForeground(Color.BLUE);

20. Call JFrame’s getContentPane method.

21. False. The java.awt and java.awt.event packages contain separate classes. To import classes from java.awt.event, you must import that package explicitly, like this:

```
import java.awt.event.*;
```

---

[1] If you declare a variable locally within createContents, you can retrieve it from an event handler by calling getSource. The getSource method is covered in Section 17.14.

[2] As an alternative, it’s legal to define an anonymous class with a superclass instead of an interface. The details of doing this are beyond the scope of this

textbook.

[3] The factorial of a number is the product of all positive integers less than or equal to that number. The factorial of n is written as n! Example: The factorial of 4 is written as 4!, and 4! is equal to 24 because 1 times 2 times 3 times 4 equals 24.

[4] In 1666, Isaac Newton discovered that white light is composed of all the colors of the color spectrum. He showed that when white light is passed through a triangular prism, it separates into different colors. And when the resulting colors are passed through a - second triangular prism, they are brought back together to form the original white light.

[5] *Look and feel* is a standard GUI term, and it refers to the appearance of something and the way in which the user interacts with it.

[6] gif stands for Graphics Interchange

Format. It’s used for an exact representation of a simple drawn image. The alternative Joint Photographic Experts Group (JPEG) format, which uses the .jpg extension, uses data compression to save storage space, but it loses information in the process.

GUI Programming—Component Layout, Additional GUI Components

Objectives

- Know GUI design basics.
- Know the benefits of using layout managers.

- Understand FlowLayout manager details.
- Understand BoxLayout manager details.
- Understand BorderLayout manager details.
- Be able to use the SwingConstants interface.
- Understand GridLayout manager details.
- Use embedded layout managers and JPanels for windows that have a substantial number of components.
- Implement JTextArea components for text that spans more than one line.
- Implement a JCheckBox component for yes/no user input.
- Implement JRadioButton and JComboBox components when the

user needs to choose a value from among a list of predefined values.

- Become familiar with additional Swing components such as menus, scroll panes, and sliders.

Outline

18.1 Introduction •••

18.2 GUI Design and Layout Managers •••

18.3 FlowLayout Manager •••

18.4 BoxLayout Manager •••

18.5 BorderLayout Manager •••

18.6 GridLayout Manager •••

18.7 Tic-Tac-Toe Example •••

18.8 Problem Solving: Winning at Tic-Tac-Toe (Optional) •••

18.9 Embedded Layout Managers •••

18.10 JPanel Class •••

18.11 MathCalculator Program •••

18.12 JTextArea Component •••

18.13 JCheckBox Component •••

18.14 JRadioButton Component •••

18.15 JComboBox Component •••

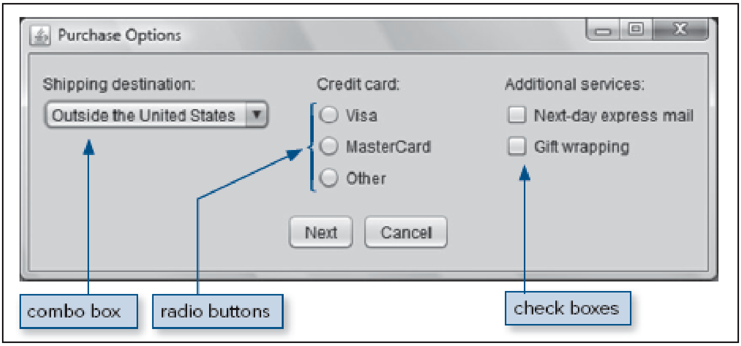
18.16 Job Application Example •••

18.17 More Swing Components •••

18.1 Introduction

This is the second chapter in our two-chapter sequence on GUI programming. In the previous chapter, you learned GUI basics. You learned about windows, components, and listeners. Almost all GUI programs need those things. In this chapter, you’ll learn how to make your GUI programs more functional and more visually appealing. You’ll improve the functionality by implementing some additional GUI components—JTextArea, JCheckBox, JRadioButton, and JComboBox. You’ll improve the visual appeal by applying various layout techniques to your windows’ components. More specifically, you’ll learn how to apply these layout managers—FlowLayout, BoxLayout, BorderLayout, and GridLayout. And you’ll learn how to apply different layout managers to different areas of your windows.

For an example of what you’ll be learning, see Figure 18.1. Note the combo box, radio button, and check box components. Also, note how the radio buttons are grouped in the center, the check buttons are grouped at the right, and the Next and Cancel buttons are grouped at the bottom center. In this chapter, you'll learn how to make such groupings, and you'll learn how to position them appropriately.



Shipping destination: Outside the United States

Purchase Options
Credit card:
○ Visa
○ MasterCard
○ Other
Additional services:
☐ Next-day express mail
☐ Gift wrapping
Next
Cancel
combo box
radio buttons
check boxes

Figure 18.1 Example window that uses radio buttons, check boxes, and a combo box

18.2 GUI Design and Layout Managers

With text-based programs, it's relatively easy to tell users what to do. As a programmer, you just provide text instructions, and the user enters input when prompted to do so. With GUI programs, it's more difficult to tell users what to do. As a programmer, you display a window with various components, set up listeners, and then wait for the user to do something. It's important that your display be easy to understand; otherwise, your users won't know what to do. To make your display easy to understand, follow these guidelines:

- Choose the right components.
- Be consistent.
- Position components appropriately.

GUI Design Basics

In Figure 18.1, note the small circles next to Visa, MasterCard, and Other. Those circles are radio button components (we describe radio buttons in Section 18.14). Using radio buttons for these credit card options is an example of choosing the right component. Radio buttons provide implicit instructions to the user about how to proceed. Most users recognize small circles as radio buttons, and when they see them, they know to click one of them with the mouse.

In Figure 18.1, note the Next and Cancel buttons at the bottom center of the window. Assume that the window is one of several windows in a purchasing application. Assume that other windows in the application also display Next and Cancel buttons in the bottom-center position. Placing Next and Cancel buttons in the same position is an example of being consistent. Consistency is important because users are more comfortable with things they've seen before. As another example, be consistent with color schemes. In a given application, if you choose red for a warning message, use red for all your warning messages.

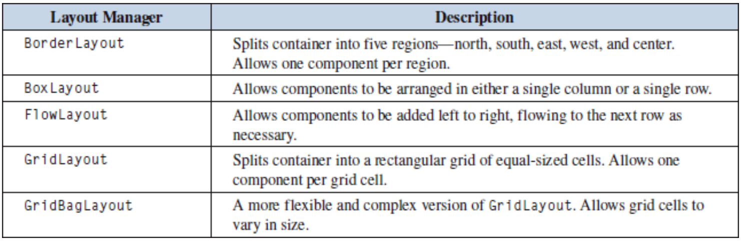
In Figure 18.1, note how the three radio button components (Visa, MasterCard, and Other) and the “Credit card:” label component are positioned together as a group. More specifically, they’re aligned in a vertical column and they’re physically close together. That’s an example of positioning components appropriately. Positioning them together as a group provides a visual cue that they’re logically related. As another example of appropriate positioning, note that there are sizable gaps separating the left, center, and right component groups. Finally, note how the “Shipping destination:”, “Credit card:”, and “Additional services:” labels are aligned in the same row. That alignment, the aforementioned gaps, and the aforementioned component groupings all lead to a more appealing and understandable display.

Layout Managers

As you now know, positioning components appropriately is an important part of GUI design. In the old days, positioning components was a tedious, manual process. Programmers would spend hours calculating the space needed for each component and the pixel coordinate positions for each component. Today, programmers are freed from that tedium by having layout managers do those calculations for them. As you may recall from the previous chapter, a *layout manager* is an object that controls the positioning of components within a container. In general, the layout manager's goal is to arrange components neatly. Usually, the neatness goal equates to making sure components are aligned and making sure components are appropriately spaced within the layout manager's container. For example, in Figure 18.1, layout managers are responsible for aligning the left components, aligning the middle components, aligning the right components, and spacing the three component groups across the width of the window.

If a user adjusts a window's size, the JVM consults with the layout manager, and the layout manager then recalculates the pixel coordinate positions for each component. All this takes place automatically, without any intervention on the programmer's part. How convenient! Hail to the layout manager!

There are different types of layout managers, and they have different strategies for positioning components within a container. See the table in Figure 18.2. It describes several layout managers from Oracle's API library.



Layout Manager	Description
`BorderLayout`	Splits container into five regions—north, south, east, west, and center. Allows one component per region.
`BoxLayout`	Allows components to be arranged in either a single column or a single row.
`FlowLayout`	Allows components to be added left to right, flowing to the next row as necessary.
`GridLayout`	Splits container into a rectangular grid of equal-sized cells. Allows one component per grid cell.
`GridBagLayout`	A more flexible and complex version of `GridLayout`. Allows grid cells to vary in size.

Figure 18.2 Several of the more popular layout managers

In the previous chapter, we used the simplest type of layout manager—the FlowLayout manager. The FlowLayout manager is useful for some situations, but we'll often need alternative layout managers for other situations. In this chapter, we'll describe the FlowLayout manager in more detail, and we'll also describe the BorderLayout, BoxLayout, and GridLayout managers. Those are the four most popular layout managers, so you should know them well.

Assigning a Layout Manager

To assign a particular layout manager to a JFrame window from within a class that extends JFrame, call the setLayout method as follows:

setLayout (new <*layout-manager-class*>(<*arguments*>));

In this code template, replace <*layout-manager-class*> by a layout manager class (like FlowLayout) and replace <*arguments*> by zero or more arguments. The arguments will be different for the different layout managers, and we'll get to those details later. If setLayout is not called, then the BorderLayout manager is used, because that's the default layout manager for a JFrame window.

The BorderLayout, FlowLayout, GridLayout, and GridbagLayout manager classes are in the java.awt package. The BoxLayout manager class is in the javax.swing package. Import accordingly.

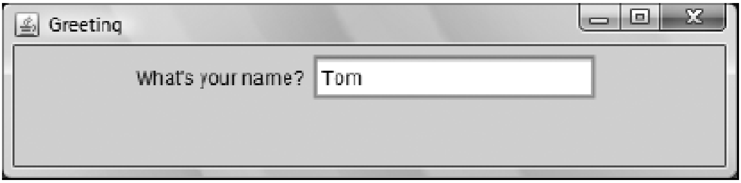
18.3 FlowLayout Manager

In the previous chapter, we wanted to present GUI basics without getting bogged down in layout manager details. So we chose a simple layout manager, FlowLayout, that didn't require much explanation. We just used it and didn't dwell on particulars. Now it's time to explain the particulars, so you can take advantage of its functionality more fully.

Layout Mechanism

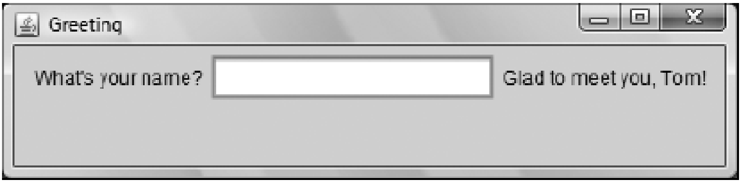
The FlowLayout class implements a simple one-compartment layout scheme that allows multiple components to be inserted into the compartment. When a component is added to the compartment, it is placed to the right of any components that were previously added to the compartment. If there is not enough room to add a component to the right of the previously added components, the new component is placed on the next line (i.e., it "flows" to the next line). Note the following example.

Assume that you've implemented a program that prompts the user to enter his or her name and prints a personalized greeting after the user presses enter. We'll show you a sample session that starts with a wide window and a short name. Here's what the program displays after the user types Tom:



What's your name? Tom

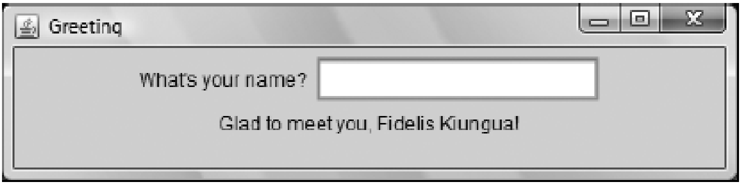
And here's what the program displays after the user presses enter:



What's your name?

Glad to meet you, Tom!

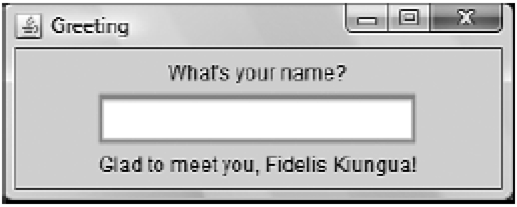
If the user enters a longer name, like Fidelis Kiungua, the greeting label can't fit on the first line, so it wraps to the next line:



What's your name?

Glad to meet you, Fidelis Kiungua!

If the user manually resizes the window to make it narrower, the text box can no longer fit on the first line, so it wraps to the next line:



What's your name?

Glad to meet you, Fidelis Kiungua!

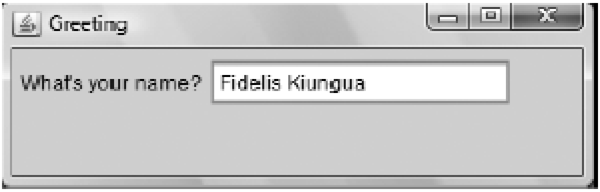
Alignment

By default, the FlowLayout manager positions its components using center alignment. For example, in the above window, note how the "What's your name?" label is centered between the left and right borders. If you'd like to change the FlowLayout manager's alignment, insert one of the

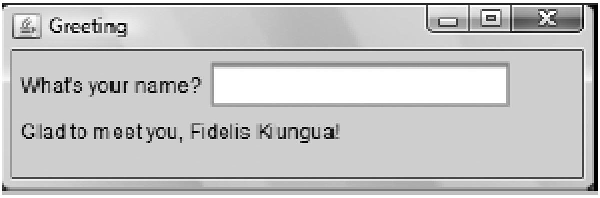
FlowLayout alignment constants (FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT) in the FlowLayout constructor call. For example, here's how to specify left alignment:

```
setLayout (new FlowLayout
(FlowLayout.LEFT));
```

Here's what our Greeting program displays when left alignment is used:



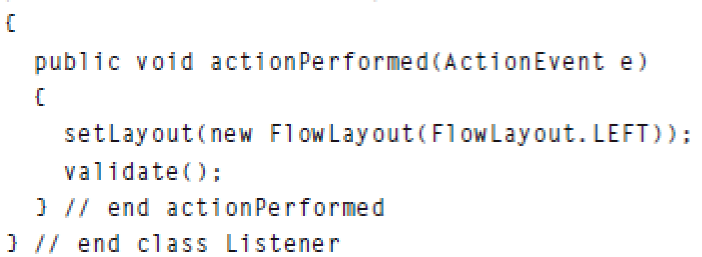
Fidelis Kiungua



Glad to meet you, Fidelis Kiungua!

Layout Changes

Normally, setLayout is called just once in a program—when the program initially lays out its components. But if there's a need to adjust the layout scheme dynamically, call setLayout again. For example, if you want the user to be able to adjust text alignment, add Align Left, Align Center, and Align Right buttons. Add a listener to each button. In each listener, call setLayout. This would be the listener for the Align Left button:



```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    setLayout(new FlowLayout(FlowLayout.LEFT));
    validate();
  } // end actionPerformed
} // end class Listener
```

Note the validate method in this code fragment. It causes the layout manager to regenerate the component layout. If you attempt to change your window's layout in some way (after the window is initially displayed with setVisible(true)) you should call validate to make sure the change takes effect. These method calls attempt to change the layout:

- setLayout—Change the window's layout manager.
- add—Add a component to the window.
- setSize—Change the window's size.
- remove—Remove a component from the window.

If your window is already visible and you call one of those methods, don't forget to call validate afterward. ⚠ If you have a series of such calls, there's no need to have separate validate method calls. Putting one validate method call at the end works fine.

18.4 BoxLayout Manager

The FlowLayout manager is popular because it's easy to use. Just add components, and the container accumulates them like words on a page of text. It starts at the top, filling each line from left to right, and then moves automatically to the next line. But suppose you want to arrange items vertically in a column. In that case, you should use a BoxLayout manager. A BoxLayout manager can arrange items in either one row or one column. If the container is wide enough, you can use a FlowLayout manager to arrange items in a row. So a BoxLayout manager is most useful when you want to arrange items in a column.

Example—a Dance Recital Poster

Suppose you want to implement a program that prints a poster with details about an upcoming dance recital. You want your program to be flexible, so it can be reused later for other dance recital events. Thus, rather than hardcoding the event details in print statements, the program prompts the user for event details, stores them, and then prints them in a pleasing format as a poster. The poster should show the following five items, one above the other:

- the name of the dance recital performance
- a promotional image, like a photograph of the featured performer
- the dance recital's date
- the dance recital's time
- the dance recital's location

Figure 18.3 shows an example of what we're talking about.

Figure 18.4 contains the dance recital program's driver, which first instantiates a DanceRecital object, where DanceRecital is the class that stores a dance recital's event details (name of the performance, image, date, time, and venue). For each of those details, the driver prompts the user to enter the relevant information and then stores it in the DanceRecital object. The driver then calls the -displayPoster method, which displays the dance recital's details using the BoxLayout manager for its layout scheme.

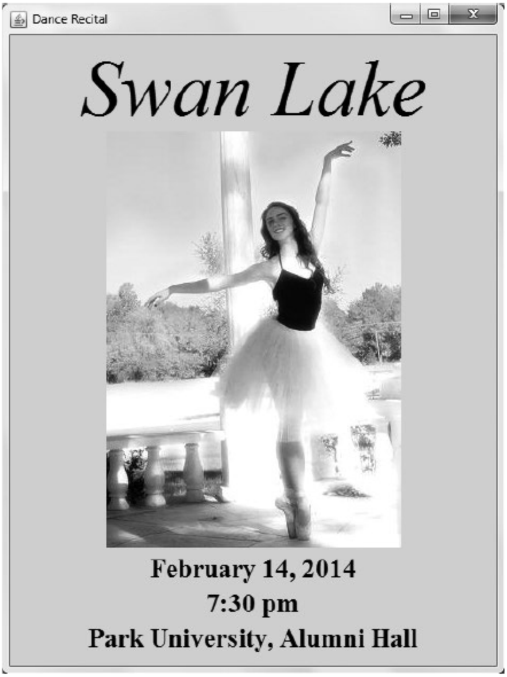
Study the DanceRecital class in Figures 18.5a and 18.5b. The first figure contains standard mutator methods, which assign dance recital details to the DanceRecital object. The second figure contains the displayPoster method, which generates a window designed to look like a dance recital poster. The programs in the previous chapter also generated windows, but they did so in a different manner. Those programs generated their windows immediately, when they called the constructor for the program's class as the first statement in main. For example, the SimpleWindow program called new SimpleWindow() as the first statement in main, and the SimpleWindow constructor was in charge of configuring the instantiated window. On the other hand, the DanceRecital program generates its window by calling the JFrame - constructor from within its displayPoster method. Here’s the relevant code, where the constructor’s argument, “Dance Recital,” specifies the window’s title:

```
JFrame frame = new JFrame("Dance Recital");
```

So why did we choose this different strategy for the DanceRecital program? For each GUI program in the last chapter, the window served as the container for input components (text boxes and buttons), and displaying the input components was essential for setting up the rest of the program’s execution. But for the DanceRecital program, the input is text based, so there’s no need to display a window right away. Actually, there might not be a need to display a window at all. In the future, a programmer might want to improve the program by adding a method that generates promotional emails for the dance recital.[1] Presumably, the programmer would rewrite the driver to allow the user to choose between displaying a poster window and generating promotional emails. By instantiating the window within the displayPoster method, we allow for that possibility.

Sample session:

```
Name of the performance: Swan Lake
Image file: dancer.jpg
Dance recital's date: February 14, 2014
Dance recital's time: 7:30 pm
Dance recital's venue: Park University, Alumni Hall
```



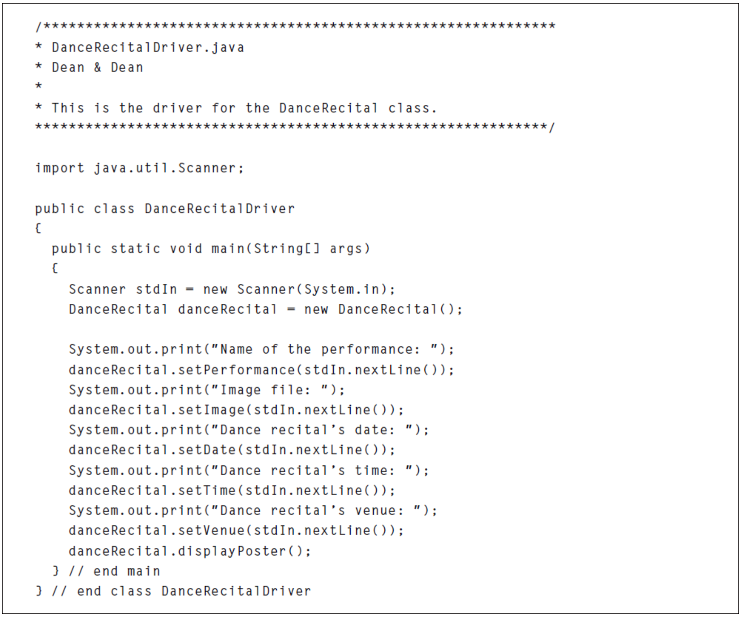
Swan Lake

February 14, 2014

7:30 pm

Park University, Alumni Hall

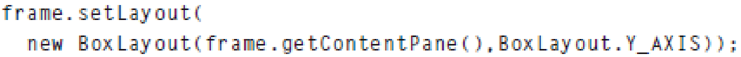
Figure 18.3 Representative output for the DanceRecital program The pictured ballerina, Jessica Sapenaro, is a dance teacher and a Park University computer science student.



```
/*************************************************************
* DanceRecitalDriver.java
* Dean & Dean
*
* This is the driver for the DanceRecital class.
*************************************************************/

import java.util.Scanner;

public class DanceRecitalDriver
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    DanceRecital danceRecital = new DanceRecital();

    System.out.print(”Name of the performance: ”);
    danceRecital.setPerformance(stdIn.nextLine());
    System.out.print(”Image file: ”);
    danceRecital.setImage(stdIn.nextLine());
    System.out.print(”Dance recital’s date: ”);
    danceRecital.setDate(stdIn.nextLine());
    System.out.print(”Dance recital’s time: ”);
    danceRecital.setTime(stdIn.nextLine());
    System.out.print(”Dance recital’s venue: ”);
    danceRecital.setVenue(stdIn.nextLine());
    danceRecital.displayPoster();
  } // end main
} // end class DanceRecitalDriver
```

Figure 18.4 Driver of the DanceRecital class in Figures 18.5a and 18.5b

We still haven’t gotten to the program’s *raison d’être* (a French expression meaning “reason for being”), which is to illustrate how to use the BoxLayout manager. The displayPoster method uses the - BoxLayout manager to display the dance recital details in a column format. To carry out that task, - displayPoster assigns a BoxLayout manager to the window by calling setLayout like this:

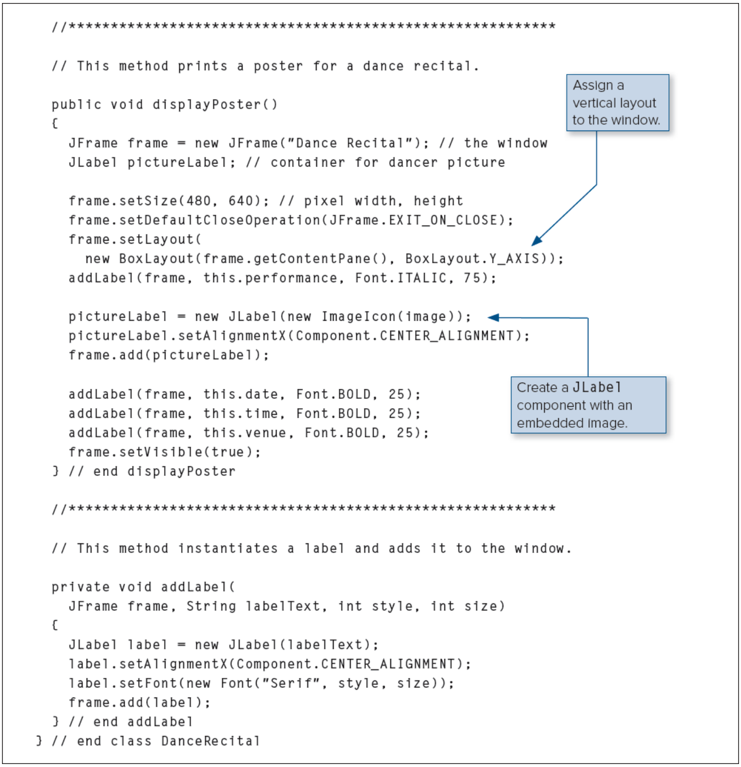


```
frame.setLayout(
  new BoxLayout(frame.getContentPane(),BoxLayout.Y_AXIS));
```

Note the embedded BoxLayout constructor call. The constructor call's first argument specifies the window's content pane (so the BoxLayout manager knows what to attach itself to). The second argument specifies the window's orientation—vertical or horizontal. To arrange all the components vertically, one on top of the other, use BoxLayout.Y_AXIS, as shown in the code above. To arrange all the components horizontally, side by side, use BoxLayout.X_AXIS.

```
/**************************************************************
* DanceRecital.java
* Dean & Dean
*
* This program stores and prints dance recital information.
**************************************************************/

import javax.swing.*;  // JFrame, JLabel, BoxLayout, ImageIcon
import java.awt.*;     // Font, Component

public class DanceRecital
{
  private String performance;  // name of the performance
  private String image;        // promotional image
  private String date;         // dance recital's date
  private String time;         // dance recital's time
  private String venue;        // dance recital's location

  //***********************************************************

  public void setPerformance(String performance)
  {
    this.performance = performance;
  } // end setPerformance

  public void setImage(String imageFile)
  {
    this.image = imageFile;
  } // end setImage

  public void setDate(String date)
  {
    this.date = date;
  } // end setDate

  public void setTime(String time)
  {
    this.time = time;
  } // end setTime

  public void setVenue(String venue)
  {
    this.venue = venue;
  } // end setVenue
```

Figure 18.5a DanceRecital class—part A



```
  //***********************************************************

  // This method prints a poster for a dance recital.

  public void displayPoster()
  {
    JFrame frame = new JFrame("Dance Recital"); // the window
    JLabel pictureLabel; // container for dancer picture

    frame.setSize(480, 640); // pixel width, height
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setLayout(
      new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS));
    addLabel(frame, this.performance, Font.ITALIC, 75);

    pictureLabel = new JLabel(new ImageIcon(image));
    pictureLabel.setAlignmentX(Component.CENTER_ALIGNMENT);
    frame.add(pictureLabel);

    addLabel(frame, this.date, Font.BOLD, 25);
    addLabel(frame, this.time, Font.BOLD, 25);
    addLabel(frame, this.venue, Font.BOLD, 25);
    frame.setVisible(true);
  } // end displayPoster

  //***********************************************************

  // This method instantiates a label and adds it to the window.

  private void addLabel(
    JFrame frame, String labelText, int style, int size)
  {
    JLabel label = new JLabel(labelText);
    label.setAlignmentX(Component.CENTER_ALIGNMENT);
    label.setFont(new Font("Serif", style, size));
    frame.add(label);
  } // end addLabel
} // end class DanceRecital
```

Assign a vertical layout to the window.

Create a `JLabel` component with an embedded image.

Figure 18.5b DanceRecital class—part B

After creating a vertical layout for the window, displayPoster adds all its components to the window. Because adding each of the poster's performance name, date, time, and venue components to the window requires much of the same code, we use a helper method, addLabel, to implement that code. As you can see in Figure 18.5b, the addLabel method creates a JLabel object, specifies center alignment, assigns the font, and finally adds the component to the window. The dancer image is a bit different than the other components. Actually, the ImageIcon is not a component at all, in that it's not a subclass of the Component class. As such, the compiler won't allow you to add it directly to the window. The trick is to first embed the ImageIcon in a JLabel object and then add the resulting JLabel object to the window. Study the displayPoster method to see how that's done.

With a FlowLayout manager, if you add more components than can fit in one row, the extra components wrap around (flow) to the next row. With a BoxLayout manager (vertical or horizontal), such wraparounds do not take place. With a BoxLayout manager window, if you add components to a full window, the new elements won't be visible. However, the components are part of the window, and the user can expose them by increasing the window's size (by maximizing or dragging a corner).

18.5 BorderLayout Manager

In this section, we discuss the BorderLayout manager, which allows you to establish the position of particular components in both horizontal and vertical dimensions.

BorderLayout Regions

The BorderLayout manager is particularly useful for windows that need components near their edges. It's common to put a title near the top edge of a window. It's common to put a menu near the left edge of a window. It's common to put buttons near the bottom edge of a window. The BorderLayout manager accommodates all those situations by splitting up its container into five *regions,* or compartments. Four of the regions are near the edges, and one is in the center. You access the four edge regions with geographical names—north, south, east, and west. Note the regions' positions in Figure 18.6.

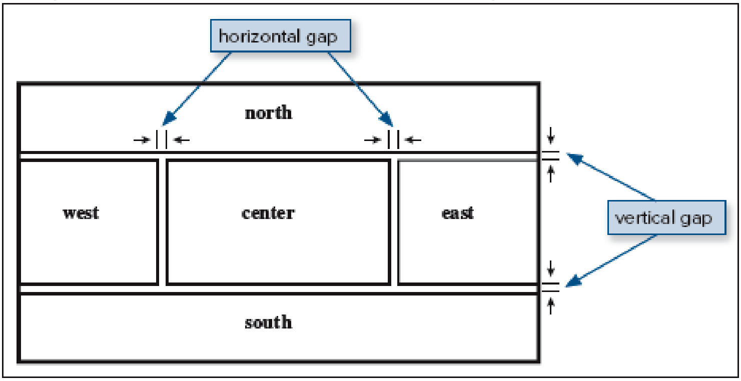


Figure 18.6 BorderLayout regions

Assume that you're inside a container class. To assign a BorderLayout manager to the container, call the setLayout method like this:

setLayout (new BorderLayout

(<*horizontal-gap*>, <*vertical-gap*>));

The *horizontal-gap* argument specifies the number of pixels of blank space that separate the west, center, and east regions. Figure 18.6 illustrates this. The *vertical-gap* argument specifies the number of pixels of blank space that separate the north region from the other regions and the south region from the other regions. Once again, Figure 18.6 illustrates this. If you omit the gap arguments, the gap values are zero by default. In other words, if you call the BorderLayout constructor with no arguments, there will be no gaps between the regions.

The sizes of the five regions are determined at runtime, and they're based on the contents of each region. Thus, if the west region contains a long label, the layout manager attempts to widen the west region. Likewise, if the west region contains a short label, the layout manager attempts to narrow the west region.

If an outer region is empty, it collapses so that it does not take up any space. But what exactly happens during the collapse? Each outer region controls only one dividing line, so only one dividing line moves for each collapsed region. Figure 18.7 shows you that the west region's dividing line is the boundary between west and center, the north region's dividing line is the boundary between north and below, and so on. So if the north region is empty, the north dividing line moves all the way up to the top border, and the west, center, and east regions all expand upward. What happens if the east and south regions are both empty? The east region being empty causes the east dividing line to move all the way to the right border. The south region being empty causes the south dividing line to move all the way down to the bottom border. Here's the resulting layout:

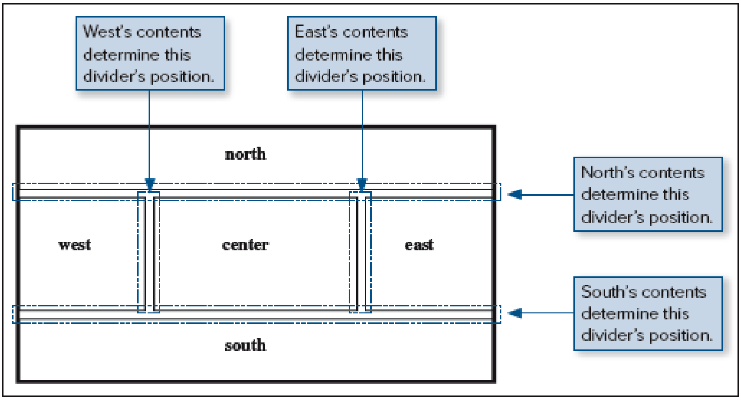
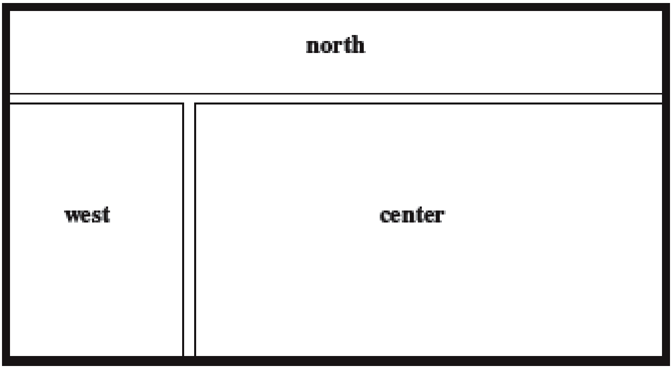

West's contents determine this divider's position.
East's contents determine this divider's position.
north
North's contents determine this divider's position.
west
center
east
South's contents determine this divider's position.
south

Figure 18.7 BorderLayout regions


north
west
center

What happens if the center region is empty? The center region doesn't control any of the dividing lines, so nothing happens.

Adding Components

Suppose you have a container class

that uses a BorderLayout manager. To add a component to one of the container's BorderLayout regions, call the container's add method like this:

add (<*component*>, <*region*>);

Replace <*component*> by a component like a JLabel or a JButton, and replace <*region*> by one of these named constants: BorderLayout.NORTH, BorderLayout.SOUTH, -BorderLayout.WEST, BorderLayout.EAST, or BorderLayout.CENTER. For example, here's how to add a Tunisia button to the north region:

add (new JButton("Tunisia"), BorderLayout.NORTH);

If you call the add method with no region argument, the center region is used by default. Thus, to add a Central African Republic button to the center region, you can use either of these two statements:

```
add(new JButton("Central African Republic"), BorderLayout.CENTER);
```

```
add(new JButton("Central African Republic"));
```

Which statement is better? We prefer the first statement because it makes the code easier to understand. More formally, we say that the first statement is *self-documenting*.

With a FlowLayout, you can add as many components as you like. With a BorderLayout, you can add only five components total, one for each of the five regions. If you add a component to a region that already has a component, then the new component replaces the old component. Thus, in executing the following lines, the Somalia button replaces the Djibouti button:

```
add(new JButton("Djibouti"), BorderLayout.EAST);
add(new JButton("Somalia ", BorderLayout.EAST));
```

If you need to add more than one component to a region, it’s easy to make the mistake of calling add twice ⚠ for the same region. After all, there’s no compile-time error to warn you of your misdeed. But what you really need to do is add a JPanel component. We’ll discuss the JPanel component later in the chapter. It allows you to store multiple components in a place where only one component is allowed.

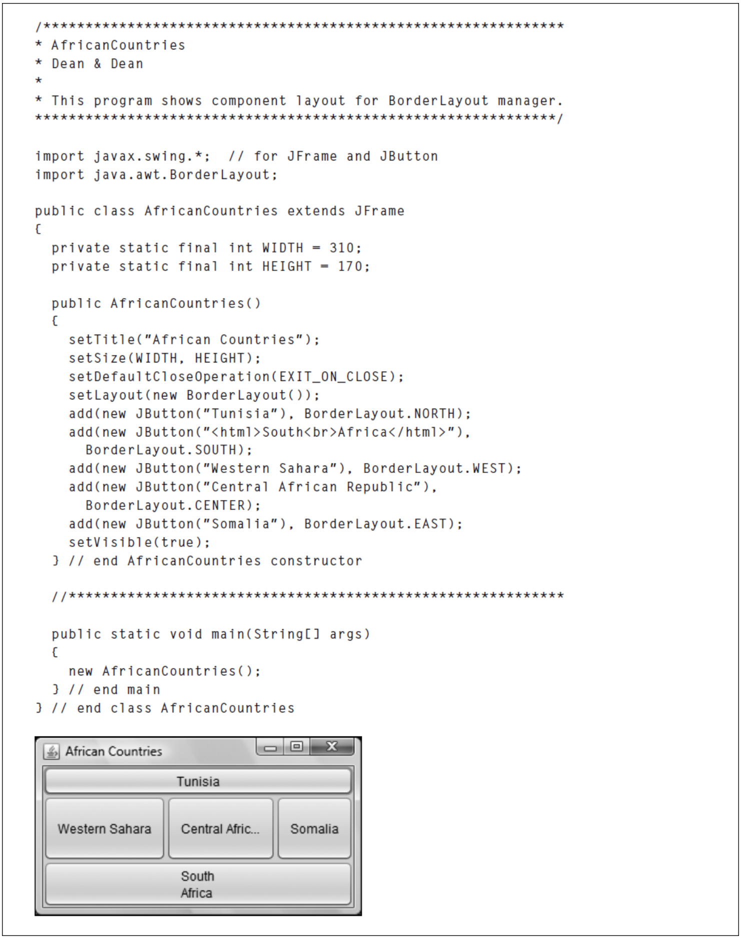
AfricanCountries Program with Buttons

OK, now let’s put this BorderLayout material into practice by using it within a complete program. In our AfricanCountries program, we add African-country buttons to the five regions of a BorderLayout window. See the program’s output window at the bottom of Figure 18.8. The five rectangles you see are the five regions, but they're also the five buttons. The buttons are the same size as the regions because, with a BorderLayout manager, components automatically expand to fill their entire region. Note how the outer four regions' sizes conform nicely to their contents. In other words, the west region is wide enough to show "Western Sahara," the south region is tall enough to show "South Africa," and so on. In contrast, note how the center region is unable to display its full "Central African Republic" content. This is because the outer regions control the dividing lines. The center region gets whatever room is left over.

Skim through the AfricanCountries program listing in Figure 18.8. Most of the code is straightforward. But this statement is rather quirky:

```
add(new
```

JButton("<html>South<br>Africa</html>"), BorderLayout.SOUTH);



```
/*******************************************************************
* AfricanCountries
* Dean & Dean
*
* This program shows component layout for BorderLayout manager.
*******************************************************************/

import javax.swing.*;  // for JFrame and JButton
import java.awt.BorderLayout;

public class AfricanCountries extends JFrame
{
  private static final int WIDTH = 310;
  private static final int HEIGHT = 170;

  public AfricanCountries()
  {
    setTitle(”African Countries”);
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());
    add(new JButton(”Tunisia”), BorderLayout.NORTH);
    add(new JButton(”<html>South<br>Africa</html>”),
      BorderLayout.SOUTH);
    add(new JButton(”Western Sahara”), BorderLayout.WEST);
    add(new JButton(”Central African Republic”),
      BorderLayout.CENTER);
    add(new JButton(”Somalia”), BorderLayout.EAST);
    setVisible(true);
  } // end AfricanCountries constructor

  //**************************************************************

  public static void main(String[] args)
  {
    new AfricanCountries();
  } // end main
} // end class AfricanCountries
```

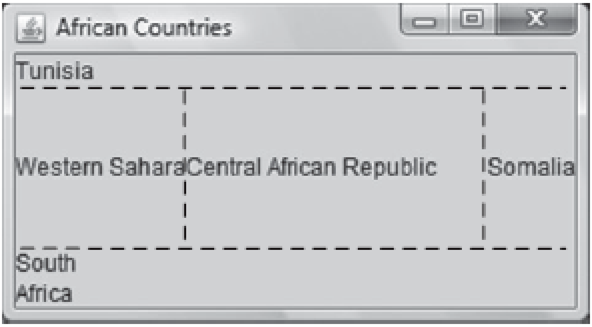
Figure 18.8 AfricanCountries program and its output

Let’s review briefly those angled-bracket commands that you see here—<html>, <br>, and </html>. As you may recall from the HTMLGenerator program in Chapter 16, the angled bracket elements are called tags. The <html> tag indicates the start of a HTML file, the <br> tag indicates a line break (i.e., a new line), and the </html> tag indicates the end of an HTML file. Normally, you insert tags into an HTML file. But here we're inserting them into component text in order to produce a new line. When used in JLabel and JButton text, the <html> and </html> tags tell the Java compiler that the enclosed text (the text between the <html> and </html> tags) should be interpreted as HTML text. And the <br> tag tells the Java compiler to insert a newline character in the text.2

We'd like to mention one additional item in the AfricanCountries program. The setLayout method call can be omitted. As we said previously, the BorderLayout is the default layout manager for JFrame windows. Therefore, if you omit the setLayout method call, the program works just fine. But we prefer to keep the setLayout call because it makes the program easier to understand.

AfricanCountries Program with Labels

You might have noticed the dividing lines in Figure 18.8’s output window. Those come from the buttons’ borders, not from the BorderLayout manager. If we used label components instead of button components, you would see no dividing lines. Likewise in Figure 18.8, the margins around the words come from the button components. If we used label components instead of button components, you would see no margins around the words. Below, we show what the AfricanCountries program displays when the button components are replaced with label components. Be aware that the dashed lines don't appear on the actual window. We've drawn them in to show you the region boundaries.


African Countries
Tunisia
Western Sahara
Central African Republic
Somalia
South
Africa

The regions are much the same as before, except that the west and east regions are narrower. That's because there are no margins around the words. Narrower west and east regions means there's more room for the center region. Thus, the center region displays its entire "Central African Republic" text.

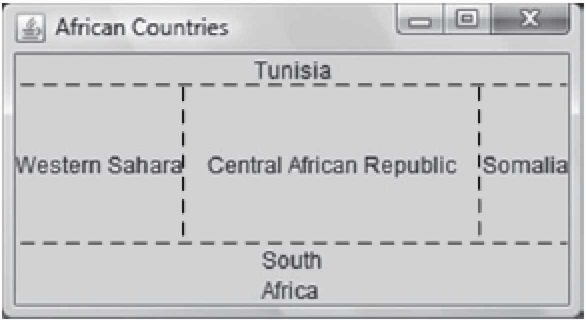
Note that the African-country labels are left aligned. That's the default for a label in a BorderLayout region. If you want a different alignment than the default, instantiate the label with an alignment constant like this:

new JLabel(<*label’s-text*>, <*alignment-constant*>)

Replace <*alignment-constant*> by one of the following named constants: SwingConstants.LEFT, SwingConstants.CENTER, or SwingConstants.RIGHT. Here’s an example that adds a center-aligned label to a BorderLayout north region:

add (new JLabel("Tunisia", SwingConstants.CENTER), BorderLayout.NORTH);

If we apply that line of code to our AfricanCountries program and we apply similar center-alignment code to our center and south regions, the program displays this:


African Countries
Tunisia
Western Sahara
Central African Republic
Somalia
South
Africa

Once again, the dashed lines don't appear on the actual window. We've drawn them to show you the region boundaries. There's no point in applying center alignment to the west and east labels. For these labels, the alignment is irrelevant because the west and east labels have no room to move. As evidenced by the dashed lines, they're already aligned with both their left and right boundaries.

Now back to the alignment constants —SwingConstants.LEFT, SwingConstants.CENTER, and SwingConstants.RIGHT. You might think that SwingConstants is a class because its first letter is capitalized. If it were a class, then it would describe an object. But it doesn't describe an object, and it's not a class. Actually, SwingConstants is an interface, defined in the javax.swing package. The Java API provides the SwingConstants interface as a repository for various GUI-related named constants. To access a named constant in the SwingConstants interface, preface the named constant with the interface name. For example, to access the LEFT alignment constant, use SwingConstants.LEFT. If you want additional details about interfaces, see Chapter 14, Section 14.9.

⚠ It's easy to get confused between label alignment for a BorderLayout container and label alignment for a FlowLayout container. With a BorderLayout container, if you want to specify a label's alignment, you need to specify a SwingConstants value as part of the label's instantiation. If you do that with a FlowLayout container, the code will compile, but it won't affect the label's alignment. With a FlowLayout container, individual component alignment is irrelevant. What matters is the container's alignment. If the container uses left alignment, then all its components are left aligned; if the container uses center alignment, then all its components are center aligned; and so on. To set the container's alignment, insert one of the FlowLayout alignment constants (FlowLayout.LEFT, FlowLayout.CENTER, FlowLayout.RIGHT) in the FlowLayout constructor call. Here's how to specify left alignment for all the components in a FlowLayout container:

```
setLayout (new FlowLayout
(FlowLayout.LEFT));
```

18.6 GridLayout Manager

The BorderLayout manager's

partitioning scheme (north, south, east, west, center) works well for many situations, but not for all situations. Often, you'll need to display information using a table format; that is, you'll need to display information that's organized by rows and columns. The BorderLayout manager doesn't work well for table formats, but the GridLayout manager works great!

GridLayout Cells

The GridLayout manager lays out a container's components in a rectangular grid. The grid is divided into equal-sized cells. Each cell can hold only one component.

Assume that you're inside a container class. To assign a GridLayout manager to the container, call the setLayout method like this:

```
setLayout(new GridLayout(<number-of-rows>, <number-of-columns>,
  <horizontal-gap>, <vertical-gap>));
```

The <*number-of-rows*> and <*number-*

of-columns> arguments specify the number of rows and number of columns, respectively, in the rectangular grid. The <*horizontal-gap*> argument specifies the number of pixels of blank space that appear between each column in the grid. The <*vertical-gap*> argument specifies the number of pixels of blank space that appear between each row in the grid. If you omit the gap arguments, the gap values are zero by default. In other words, if you call the GridLayout constructor with only two arguments, there will be no gaps between the cells.

Adding Components

Assume that you're inside a GridLayout container class. To add a component to one of the container's cells, call the add method like this:

add(<*component*>);

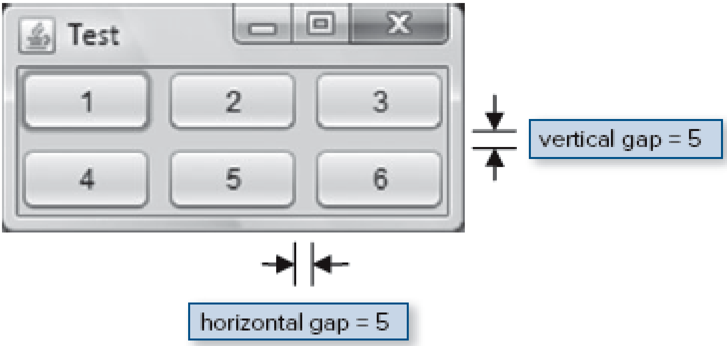
Note the simplicity of the add method

call. In particular, note that there’s no mention of the cell that the component plugs into. So how does the GridLayout manager know which cell to plug the component into? The GridLayout manager positions components within the container using left-to-right, top-to-bottom order. The first added component goes in the top-left-corner cell, the next added component goes in the cell to the right of the first component, and so on.

The code fragment below generates a two-row, three-column table with six buttons. The code fragment specifies gaps of 5 pixels between the rows and columns.

```
setLayout(new GridLayout(2, 3, 5, 5));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
add(new JButton("6"));
```

Assume that the above code fragment is part of a complete, working program. Here’s what the program displays:



The six rectangles you see are the six cells, but they’re also the six buttons. The buttons are the same size as the cells because, with a GridLayout manager, components expand to fill their cells. That should sound familiar; BorderLayout components do the same thing.

Specifying Number of Rows and Number of Columns

When creating a GridLayout manager, you call the GridLayout constructor with a number-of-rows argument and a number-of-columns argument. Those two arguments require some explanation. To help with the explanation, consider three different cases.

Case one:

If you know the number of rows and columns in your table and the table will be completely filled in (i.e., there are no empty cells), call the GridLayout constructor with the actual number of rows and the actual number of columns. That's what we did in our previous example. We knew we wanted a two-row by three-column table with six buttons, so we specified 2 for the rows argument and 3 for the columns argument.

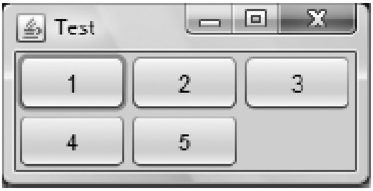
Case two:

Sometimes you might want a row-oriented display. In other words, you want a certain number of rows displayed, and you don't care about or aren't sure about the number of columns. If that's the case, call the GridLayout constructor with the actual number of rows for the rows argument and 0 for the columns argument. A 0 for the columns argument indicates that you're leaving it up to the GridLayout manager to determine the number of columns.

The code fragment below generates a two-row GridLayout with five buttons. Because the setLayout call does not specify gap values, the GridLayout displays no gaps between the buttons.

```
setLayout(new GridLayout(2, 0));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
```

Assume the above code fragment is part of a complete, working program. Here's what the program displays:



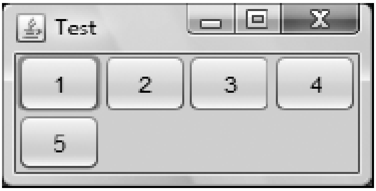
Case three:

Sometimes you might want a column-oriented display. In other words, you want a certain number of columns displayed, and you don't care about or aren't sure about the number of rows. If this is the case, call the GridLayout constructor with the actual number of columns for the columns argument and 0 for the rows argument. A 0 for the rows argument indicates that you're leaving it up to the GridLayout manager to determine the number of rows.

The code fragment below generates a four-column GridLayout with five buttons:

```
setLayout(new GridLayout(0, 4));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
add(new JButton("4"));
add(new JButton("5"));
```

Assume that the above code fragment is part of a complete, working program. Here's what the program displays:

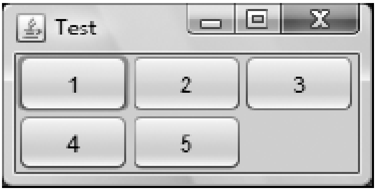


Now for a couple of things to watch for. As you know, there's special significance when you call the GridLayout constructor with rows = 0 or columns = 0. It puts the GridLayout manager in charge of choosing the number of rows or the number of columns. But it only works if you have one 0-value argument, not two. If you call the GridLayout constructor with two 0-value arguments, you'll get ⚠ a run-time error.

What about the opposite case—when you call the GridLayout constructor with two non-0 values for the rows and columns arguments. That's fine, as long as your table is completely filled. If it's not completely filled, you might get unexpected results. For example, the above four-column window is not ⚠-completely filled. Suppose you accidentally specify a value for the rows argument:

```
setLayout(new GridLayout(2, 4));
```

Here's what the program displays:



Now that's strange! There are three columns, even though we specified four. Moral of the story: Call the GridLayout constructor with two non-0 values only if your table is completely filled.[3]

18.7 Tic-Tac-Toe Example

In this section, we present a simple tic-tac-toe program. We've chosen tic-tac-toe because we wanted to illustrate GridLayout details. And tic-tac-toe, with its three-row by three-column board, provides the perfect opportunity for that.

User Interface

The program initially displays a three-row, three-column grid of blank buttons. Two users, player X and player O, take turns clicking blank buttons. Player X goes first. When player X clicks a button, the button's label changes from blank to X. When player O clicks a button, the button's label changes from blank to O. Player X wins by getting three X's in a row, 3 X's in a column, or 3 X's in a diagonal. Player O wins in the same manner, except that O's are looked at instead of X's. To get a better handle on all this, see the sample session in Figure 18.9.

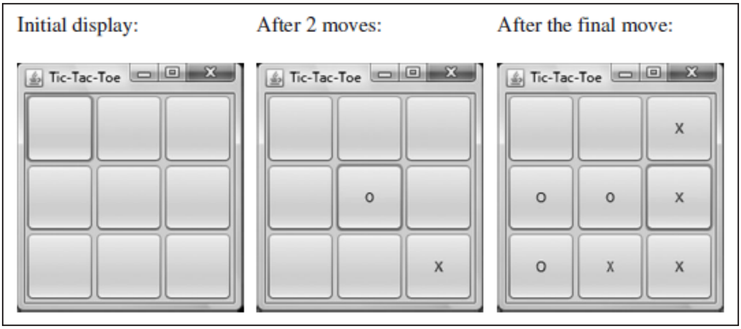

Initial display:
After 2 moves:
After the final move:
Tic-Tac-Toe
Tic-Tac-Toe
Tic-Tac-Toe
O
X
X
O
O
X
O
X
X

Figure 18.9 Sample session for the TicTacToe program

Program Details

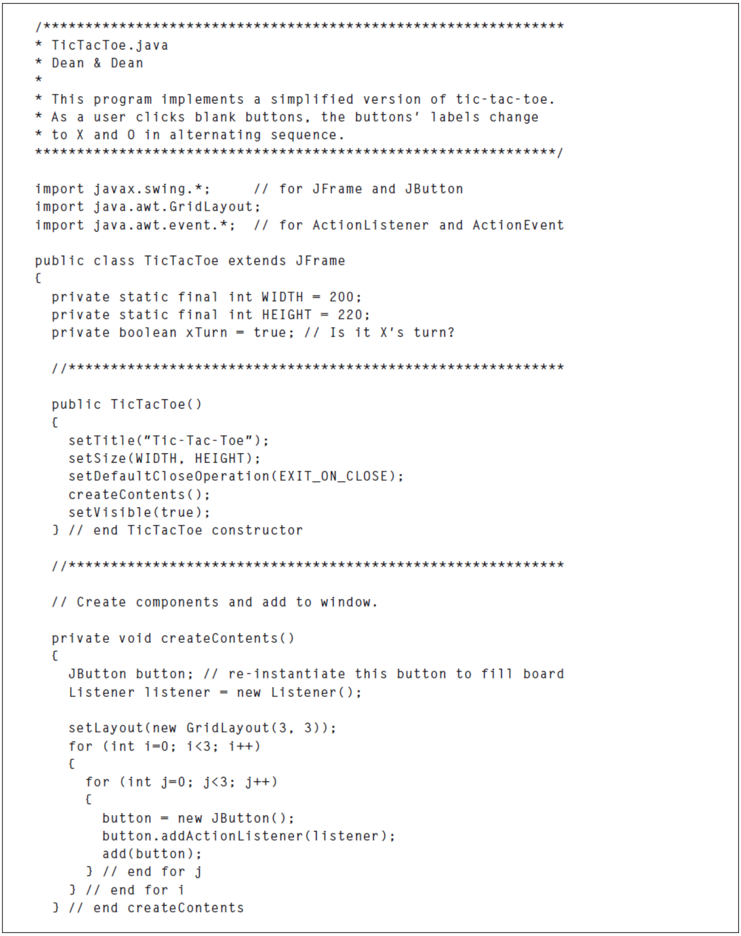
See the TicTacToe program listing in Figures 18.10a and 18.10b. Most of the code should make because already because its structure parallels the structure in our previous GUI programs. We'll skip the familiar code and focus on the more difficult code.

Note the setLayout method call in Figure 18.10a. It contains a GridLayout constructor call that specifies three rows and three columns. The constructor call does not include horizontal-gap and vertical-gap arguments, so the tic-tac-toe buttons display with no gaps between them.

Now let’s take a look at the Listener class in Figure 18.10b. In particular, note the statement where we get the clicked button and save it in a local variable:

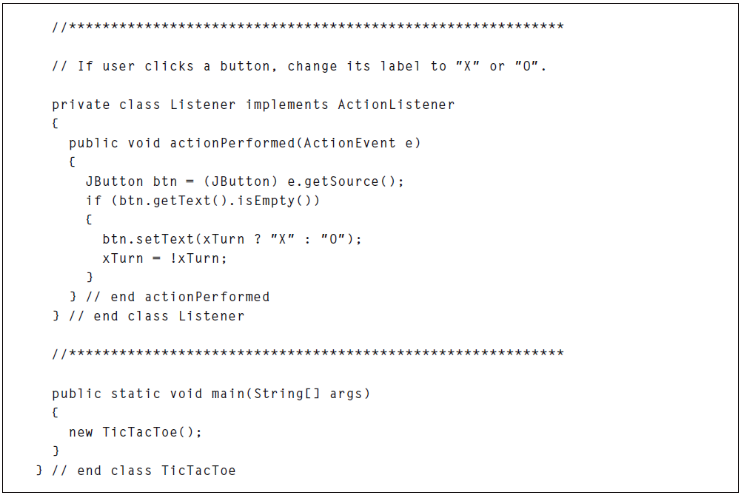
```
JButton btn = (JButton) e.getSource();
```

⚠ The (JButton) cast operator is used because if there were no cast operator, the compiler would generate an error. Why? Because the compiler would see an Object at the right being assigned to a JButton at the left (it sees an Object at the right because getSource is defined with an Object return type). In this case, because getSource really returns a JButton, it’s legal to cast its returned value to JButton, and that satisfies the compiler and eliminates the error.



```
/**************************************************************
* TicTacToe.java
* Dean & Dean
*
* This program implements a simplified version of tic-tac-toe.
* As a user clicks blank buttons, the buttons’ labels change
* to X and O in alternating sequence.
**************************************************************/

import javax.swing.*;     // for JFrame and JButton
import java.awt.GridLayout;
import java.awt.event.*;  // for ActionListener and ActionEvent

public class TicTacToe extends JFrame
{
  private static final int WIDTH = 200;
  private static final int HEIGHT = 220;
  private boolean xTurn = true; // Is it X's turn?

  //***********************************************************

  public TicTacToe()
  {
    setTitle(“Tic-Tac-Toe”);
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end TicTacToe constructor

  //***********************************************************

  // Create components and add to window.

  private void createContents()
  {
    JButton button; // re-instantiate this button to fill board
    Listener listener = new Listener();

    setLayout(new GridLayout(3, 3));
    for (int i=0; i<3; i++)
    {
      for (int j=0; j<3; j++)
      {
        button = new JButton();
        button.addActionListener(listener);
        add(button);
      } // end for j
    } // end for i
  } // end createContents
```

Figure 18.10a TicTacToe program—part A



```
  //*************************************************************

  // If user clicks a button, change its label to "X" or "O".

  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      JButton btn = (JButton) e.getSource();
      if (btn.getText().isEmpty())
      {
        btn.setText(xTurn ? "X" : "O");
        xTurn = !xTurn;
      }
    } // end actionPerformed
  } // end class Listener

  //*************************************************************

  public static void main(String[] args)
  {
    new TicTacToe();
  }
} // end class TicTacToe
```

Figure 18.10b TicTacToe program—part B

Let’s examine the Listener class’s if statement:

```
if (btn.getText().isEmpty())
{
  btn.setText(xTurn ? "X" : "O");
  xTurn = !xTurn;
}
```

We first check to ensure that the button is a blank button. We then reassign the button’s label by using a conditional operator. If xTurn holds true, then X is assigned to the button label. Otherwise, O is assigned to the button label. We then change the value of xTurn by assigning its negated value into it. More specifically, if xTurn is false, we assign true to xTurn. And if xTurn is true, we assign false to xTurn.

18.8 Problem Solving: Winning at Tic-Tac-Toe (Optional)

Iterate to enhance.

As you might have noticed, the previous section’s TicTacToe program doesn’t check for a winning move. As a problem-solving exercise, let’s now discuss how to add that functionality. Rather than provide you with a Java solution, we’ll provide you with the thought process for coming up with a solution. We’ll codify the thought process using pseudocode. One of this chapter’s projects asks you to finish the job by implementing a complete Java program solution.

To check for a win (i.e., to check for three in a row, three in a column, or three in a diagonal), the listener needs to access multiple buttons. As it stands now, the TicTacToe listener can access only one button—the button that was clicked. It gets that button by calling getSource. So how should you change the program so the listener can access multiple buttons?

To access multiple buttons, you need to declare multiple buttons. You could declare nine separate buttons, but the more elegant solution is to declare a three-row, three-column, two-dimensional array of buttons. The next question is, where should you declare the array? Do you declare it as a local variable inside the listener or as an instance variable at the top of the program? In general, local variables are preferred, but in this case, a local variable won't work. Local variables don't persist. You need to be able to update a button from within the listener and have that update be remembered the next time the listener is called. Thus, you need to declare the buttons array as an instance variable.

You need to check for a win only when the user clicks a button. So add check-for-a-win code to the actionPerformed method inside the button's listener. In adding the code, use top-down design. In other words, don't worry about the low-level details; just assume they work. Here's the updated actionPerformed method. The added code is in pseudocode:

```
public void actionPerformed(ActionEvent e)
{
  JButton btn = (JButton) e.getSource();
  if (btn.getText().isEmpty())
  {
    btn.setText(xTurn ? "X" : "O");
    if there's a win
    {
      print winning player
      prepare for new game
    }
    else
    {
      xTurn = !xTurn;
    }
  }
} // end actionPerformed
```

pseudocode

The pseudocode contains three tasks—checking for a win, printing the winner, and preparing for a new game. Checking for a win requires the most thought, so we'll postpone that task for now. Let's discuss the other two tasks first.

Printing the winner should be straightforward. Just call JOptionPane.showMessageDialog with a congratulatory message. The message should include the player's name, X or O, which can be obtained by re-using the conditional operator code, xTurn ? "X" : "O".

Preparing for a new game should be straightforward as well. Just assign the empty string to the board's button labels and assign true to the xTurn variable (X always goes first).

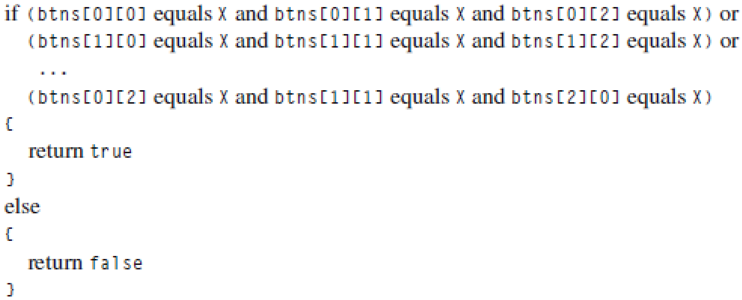
Feel free to implement the print-winning-player and prepare-for-new-game tasks as embedded code inside the if statement or as separate helper methods. Either way is fine. But the checking-for-a-win task definitely should be implemented as a separate helper method. Why? Note how cleanly win is called in the above pseudocode. You can retain that clean look in the final Java code only if you implement the checking- for-a-win task as a method, not as embedded code.

In implementing the win method, you need to check the two-dimensional buttons array for three in a row, three in a column, or three in a diagonal.

Normally, when you access a group of elements in an array, you should use a for loop. So you might want to use a for loop to access the elements in the first row, use another for loop to access the elements in the second row, and so on. But that would require eight for loops:

for loop to handle first row
for loop to handle second row
...
for loop to handle second diagonal

Yikes! That's a lot of for loops! Is there a better way? How about taking the opposite approach and using no for loops? Use one big if statement like this:



```
if (btns[0][0] equals X and btns[0][1] equals X and btns[0][2] equals X) or
  (btns[1][0] equals X and btns[1][1] equals X and btns[1][2] equals X) or
   ...
  (btns[0][2] equals X and btns[1][1] equals X and btns[2][0] equals X)
{
  return true
}
else
{
  return false
}
```

That works fine, but if you're bothered

by the length of the if condition (eight lines long), you might want to try the following. Use one for loop for all the rows, one for loop for all the columns, and one if statement for the two diagonals:

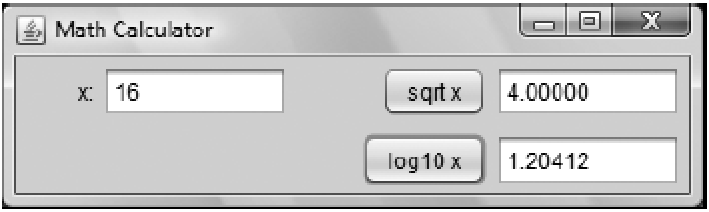
```
for (i←0; i<3; i++)
{
  if (btns[i][0] equals X and btns[i][1] equals X and btns[i][2] equals X)
  {
    return true
  }
}
for (j←0; j<3; j++)
{
  if (btns[0][j] equals X and btns[1][j] equals X and btns[2][j] equals X)
  {
    return true
  }
}
if (btns[0][0] equals X and btns[1][1] equals X and btns[2][2] equals X) or
  (btns[0][2] equals X and btns[1][1] equals X and btns[2][0] equals X)
{
  return true
}
return false
```

Of the three solutions, we prefer the last one because we feel its code is the most understandable.

To make the tic-tac-toe program more "real world," you'd probably want to provide additional functionality. In particular, you'd want to check for a "cat's game," which is when the board is filled and no one has won. You’re asked to implement that functionality in one of the chapter’s projects.

18.9 Embedded Layout Managers

Suppose that you’d like to implement this math-calculator window:



x: 16

4.00000

1.20412

What type of layout scheme should you use? Coming up with a good layout scheme often requires - creativity. We’ll walk you through the creative process for this math-calculator example.

Trying Out the Different Layout Managers

The math-calculator window appears to have two rows and four columns. So is a two-row by four-column Gridlayout scheme appropriate? The GridLayout manager is usually adequate for positioning components in an organized, tabular fashion, but it's limited by one factor—each of its cells must be the same size. If we use a two-row by four-column GridLayout scheme for the math-calculator window, then we'll have eight same-sized cells. That's fine for most of the cells, but not for the top-left cell. The top-left cell would hold the x: label. With such a small label, we would want a relatively small cell for it. But with a GridLayout scheme, a "relatively small cell" is not an option.

Because the GridLayout manager is less than ideal, you might want to think about the FlowLayout manager. That could sort of work if you use right-aligned components. But then you'd be at the mercy of the user not resizing the window. If the user widens the window, then the log10 x button would flow up to the top line, and you don't want that. So the FlowLayout manager is also less than ideal. The BorderLayout manager isn't even close. So what's the solution?

Using an Embedded Layout Scheme

Delegate.

In coming up with layouts for more complex windows, the key is often to embed layout managers inside other layout managers. Let's first tackle the outer layout manager. For the math calculator window, we want the input at the left and the output at the right. Those two entities are approximately the same width, so it makes sense to consider using a two-column GridLayout for them. The left column would contain the input components—the x label and the input text box. The right column would contain the output components—the square root's button and output text box and the logarithm's button and output text box. We'd like to organize the output components so that the square root's items are above the logarithm's items. That means using two rows for our GridLayout. See the two-row by two-column GridLayout scheme in Figure 18.11.

As you know, GridLayout managers only allow one component per cell. But Figure 18.11 shows two components in the top-left cell and two components in the top-right cell. To implement that organization scheme, you'll need to group each of the two-component pairs into their own separate containers. And to achieve the proper layout, you'll need to apply layout managers to each of those containers. The top-left cell's container uses a center-aligned FlowLayout manager. The right cells' containers use right-aligned FlowLayout managers. Voilà—layout managers inside a layout manager. Pretty cool, eh?

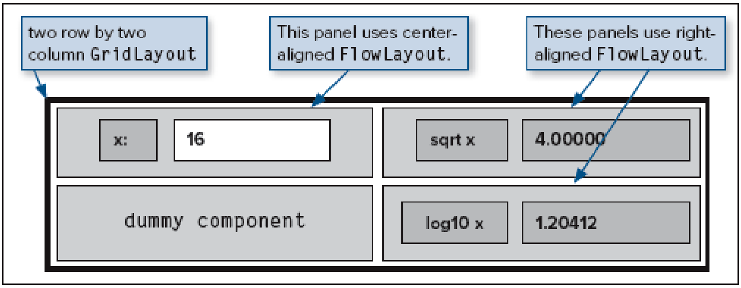

two row by two column GridLayout
This panel uses center-aligned FlowLayout.
These panels use right-aligned FlowLayout.
x:
16
sqrt x
4.00000
dummy component
log10 x
1.20412

Figure 18.11 GridLayout with embedded FlowLayout panels in three of the cells

When you have a non-trivial window, it's very common to have embedded layout managers. And when that happens, it can take a considerable amount of tweaking to get your windows to look right. Despite the tweaking, using embedded layout managers is still a lot easier than having to position components manually with pixel values like in the old days. The next section provides details on the containers for the embedded layout managers.

18.10 JPanel class

Before continuing with the implementation of the math-calculator program, we need to discuss the JPanel class. A JPanel container object is a generic storage area for components. If you have a complicated window with many components, you might want to compartmentalize the components by putting groups of components in JPanel containers. JPanel containers are particularly useful with GridLayout and BorderLayout windows because each compartment in those layouts can store only one component. If you need a compartment to store more than one component, let that one component be a JPanel container, and put multiple components into the JPanel container.

Implementation

As you may recall, GUI classes that begin with *J* come from the javax.swing package. So that's where the JPanel container class comes from, and you need to import the javax.swing package in order to use Jpanel.

To instantiate a JPanel container, use this syntax:

JPanel <JPanel-*reference*> = new JPanel(<*layout-manager?*>);

The *layout-manager* argument is optional. If it's omitted, the default is a center-aligned FlowLayout manager.

So the JPanel container's default layout manager is FlowLayout. Quick quiz: Do you remember the JFrame container's default layout manager? It's BorderLayout. That should make sense when you realize that JFrame containers are designed to handle the window as a whole. For the window as a whole, the default BorderLayout scheme works well because its report-oriented regions (north for a header, south for a footer, center for a main body) match the needs of many program windows. On the other hand, JPanel containers are designed for compartments within a window. For such compartments, the default FlowLayout scheme works well because its free-form flow matches the needs for many compartments.

Adding Components to a JPanel

After instantiating a JPanel, you'll want to add components to it. Adding components to a JPanel is the same as adding components to a JFrame. Call the add method. As you know, the add method works differently for the different layout managers. If your JPanel uses either a FlowLayout manager or a GridLayout manager, call the add method like this:

<JPanel-*reference*>.add(<*component*>);

If your JPanel uses a BorderLayout manager, you should add a second argument to specify the component's region:

<*JPanel-reference*>.add(<*component*>, <*BorderLayout-region*>);

Adding JPanel to a Window

After adding components to a JPanel, you'll need to add the JPanel to a window. If your window uses a FlowLayout manager or a GridLayout manager, call the add method like this:

add(<*JPanel-reference*>);

If your window uses a BorderLayout manager, you'll want to add a second argument to specify the component's region:

add(<*JPanel-reference*>,

```
<BorderLayout-region>);
```

In the next section, we return to the math-calculator program. That will give us an opportunity to see how JPanel works in the context of a complete program.

18.11 MathCalculator Program

See the MathCalculator program listing in Figures 18.12a, 18.12b, and 18.12c. You should peruse the entire program on your own, but we'll focus primarily on the panel-related code.

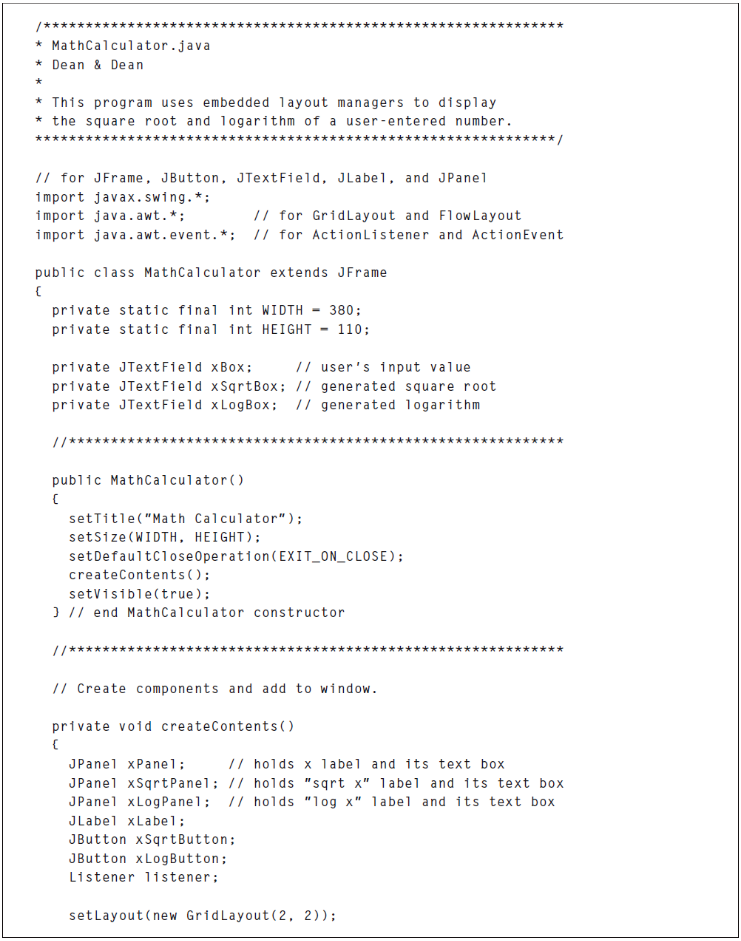
From the MathCalculator program's createContents method, here's the code that creates the top-left cell's panel:

```
xPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
xPanel.add(xLabel);
xPanel.add(xBox);
```

The first statement instantiates the JPanel container. Because the JPanel constructor uses a center-aligned FlowLayout by default, you can write the first statement like this and get the same result:

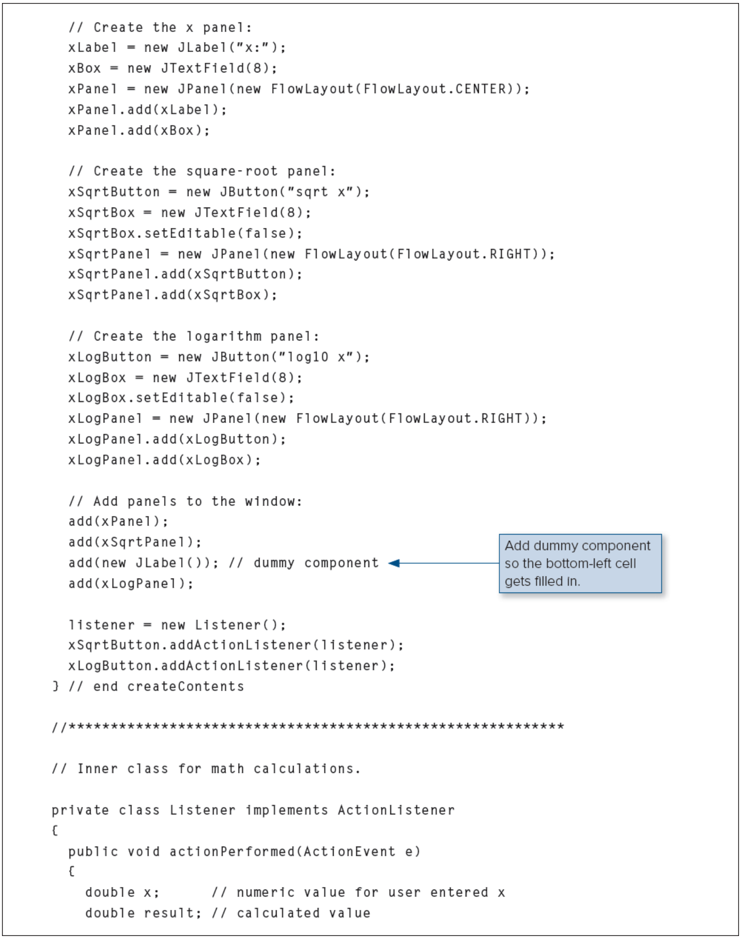
```
xPanel = new JPanel();
```

But we prefer the original statement because it's self-documenting. The second and third statements add the x: label and the input text box to the panel.



```
/**************************************************************
* MathCalculator.java
* Dean & Dean
*
* This program uses embedded layout managers to display
* the square root and logarithm of a user-entered number.
**************************************************************/

// for JFrame, JButton, JTextField, JLabel, and JPanel
import javax.swing.*;
import java.awt.*;        // for GridLayout and FlowLayout
import java.awt.event.*;  // for ActionListener and ActionEvent

public class MathCalculator extends JFrame
{
  private static final int WIDTH = 380;
  private static final int HEIGHT = 110;

  private JTextField xBox;     // user's input value
  private JTextField xSqrtBox; // generated square root
  private JTextField xLogBox;  // generated logarithm

  //***********************************************************

  public MathCalculator()
  {
    setTitle(”Math Calculator”);
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end MathCalculator constructor

  //***********************************************************

  // Create components and add to window.

  private void createContents()
  {
    JPanel xPanel;     // holds x label and its text box
    JPanel xSqrtPanel; // holds ”sqrt x” label and its text box
    JPanel xLogPanel;  // holds ”log x” label and its text box
    JLabel xLabel;
    JButton xSqrtButton;
    JButton xLogButton;
    Listener listener;

    setLayout(new GridLayout(2, 2));
```

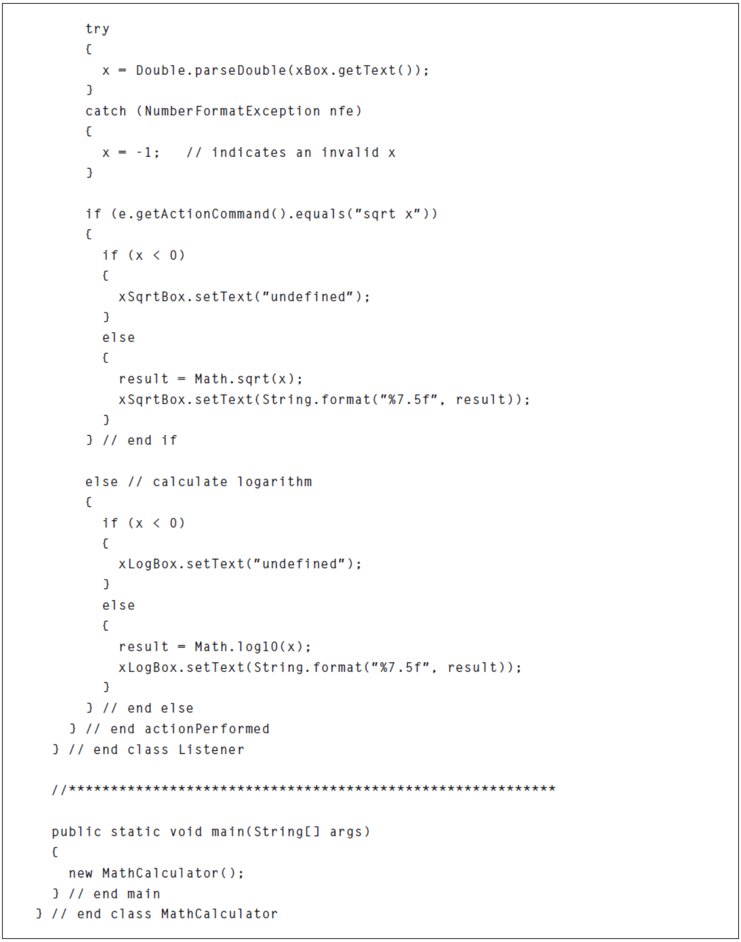
Figure 18.12a MathCalculator program—part A



```
    // Create the x panel:
    xLabel = new JLabel(”x:”);
    xBox = new JTextField(8);
    xPanel = new JPanel(new FlowLayout(FlowLayout.CENTER));
    xPanel.add(xLabel);
    xPanel.add(xBox);

    // Create the square-root panel:
    xSqrtButton = new JButton(”sqrt x”);
    xSqrtBox = new JTextField(8);
    xSqrtBox.setEditable(false);
    xSqrtPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    xSqrtPanel.add(xSqrtButton);
    xSqrtPanel.add(xSqrtBox);

    // Create the logarithm panel:
    xLogButton = new JButton(”log10 x”);
    xLogBox = new JTextField(8);
    xLogBox.setEditable(false);
    xLogPanel = new JPanel(new FlowLayout(FlowLayout.RIGHT));
    xLogPanel.add(xLogButton);
    xLogPanel.add(xLogBox);

    // Add panels to the window:
    add(xPanel);
    add(xSqrtPanel);
    add(new JLabel()); // dummy component
    add(xLogPanel);

    listener = new Listener();
    xSqrtButton.addActionListener(listener);
    xLogButton.addActionListener(listener);
  } // end createContents

  //***********************************************************

  // Inner class for math calculations.

  private class Listener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      double x;      // numeric value for user entered x
      double result; // calculated value
```

Add dummy component so the bottom-left cell gets filled in.

Figure 18.12b MathCalculator program—part B



```
      try
      {
        x = Double.parseDouble(xBox.getText());
      }
      catch (NumberFormatException nfe)
      {
        x = -1;   // indicates an invalid x
      }

      if (e.getActionCommand().equals("sqrt x"))
      {
        if (x < 0)
        {
          xSqrtBox.setText("undefined");
        }
        else
        {
          result = Math.sqrt(x);
          xSqrtBox.setText(String.format("%7.5f", result));
        }
      } // end if

      else // calculate logarithm
      {
        if (x < 0)
        {
          xLogBox.setText("undefined");
        }
        else
        {
          result = Math.log10(x);
          xLogBox.setText(String.format("%7.5f", result));
        }
      } // end else
    } // end actionPerformed
  } // end class Listener

  //**********************************************************

  public static void main(String[] args)
  {
    new MathCalculator();
  } // end main
} // end class MathCalculator
```

Figure 18.12c MathCalculator program—part C

Farther down in the createContents method, here's the code that adds the panels to the window:

```
add(xPanel);
add(xSqrtPanel);
add(new JLabel()); // dummy component
add(xLogPanel);
```

The first, second, and fourth statements add the three panels to the window’s top-left, top-right, and bottom-right cells, respectively. The third statement adds a dummy component (a blank label) to the bottom-left cell. The dummy component is necessary because without it, the xLogPanel would go into the bottom- left cell, and that’s not what you want.

There’s one additional item worth mentioning in this program. Note the String.format method call in Figure 18.12c’s actionPerformed method. The String.format method works the same as the printf method, except that instead of printing a formatted value, it returns a formatted value. In the actionPerformed method, we call String.format to retrieve a formatted version of the calculated logarithm value. Specifically, the %7.5f conversion specifier returns a floating-point value with 5 decimal places and 7 total characters.

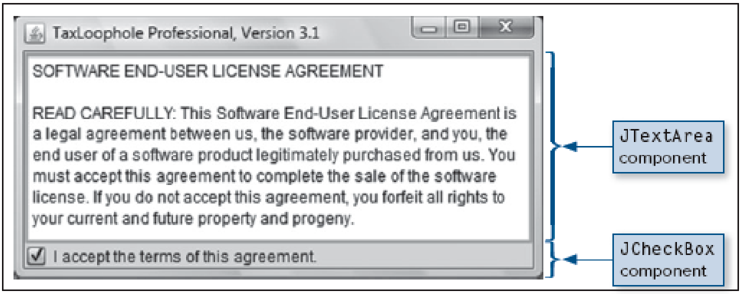
18.12 JTextArea Component

In the previous chapter, we introduced you to a few GUI components—JLabel, JTextField, JButton, and JOptionPane—that provide basic input/output functionality. Now we'll introduce you to a few more GUI components—JTextArea, JCheckBox, JRadioButton, and JComboBox—that provide more advanced input/output functionality. We'll start with the JTextArea component.

User Interface

The JLabel component works great for

displaying a single line of text. You can use a JLabel component to display multiple lines of text, but achieving multiple lines requires cluttering up your code with HTML tags—<html> … <br />…</html>. The preferred technique for displaying multiple lines of text is to use a JTextArea component. The large white area in Figure 18.13 is a JTextArea component. By the way, the small shaded area at the bottom of the figure is a JCheckBox component. We'll describe JCheckBox components in the next section.


TaxLoophole Professional, Version 3.1

SOFTWARE END-USER LICENSE AGREEMENT

READ CAREFULLY: This Software End-User License Agreement is a legal agreement between us, the software provider, and you, the end user of a software product legitimately purchased from us. You must accept this agreement to complete the sale of the software license. If you do not accept this agreement, you forfeit all rights to your current and future property and progeny.

☑ I accept the terms of this agreement.

Figure 18.13 A window with a JTextArea component and a JCheckBox component

Implementation

To create a JTextArea component, call the JTextArea constructor like this:

JTextArea <*JTextArea-reference*> = new JTextArea(<*display-text*>);

The *display-text* is the text that appears in the JTextArea component. If the display-text argument is omitted, then the JTextArea component displays no initial text.

Methods

The JTextArea class, like all the GUI component classes, has quite a few methods. Here are the API headings and descriptions for the more popular JTextArea methods:

public String getText()

Returns the text area’s text.

public void setText(String text)

Assigns the text area’s text.

public void setEditable(boolean flag)

Makes the text box editable or non-editable.

public void setLineWrap(boolean flag)

Turns line wrap on or off.

public void setWrapStyleWord(boolean flag)

Specifies whether word boundaries are used for line wrapping.

JTextArea components are editable by default, which means users are allowed to type inside them. If you want to prevent users from editing a JTextArea component, call setEditable with an argument value of false. Doing so prevents users from updating the text area, but it does not prevent programmers from updating the text area. Programmers can call the setText method regardless of whether the text area is editable or non-editable.

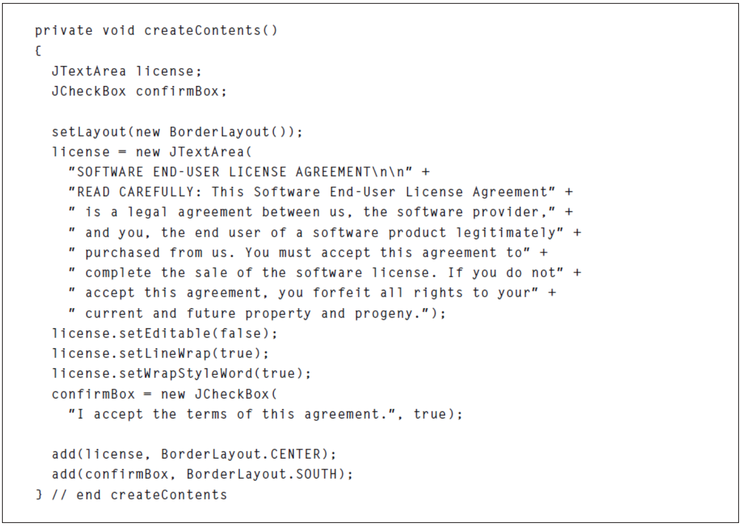
JTextArea components have line wrap turned off by default. Normally, you'll want to turn line wrap on by calling setLineWrap(true). That way, long lines automatically wrap to the next row, instead of disappearing when they reach the text area's right boundary.

For JTextArea components with line wrap turned on, the default is to perform line wrap at the point where the text meets the text area's right boundary, regardless of whether that point is in the middle of a word. Normally, you'll want to avoid that draconian4 default behavior and have line wrap occur only at word boundaries. To change to a word-boundary line-wrap policy, call setWrapStyleWord(true).

License-Agreement Example

Look back at the license-agreement

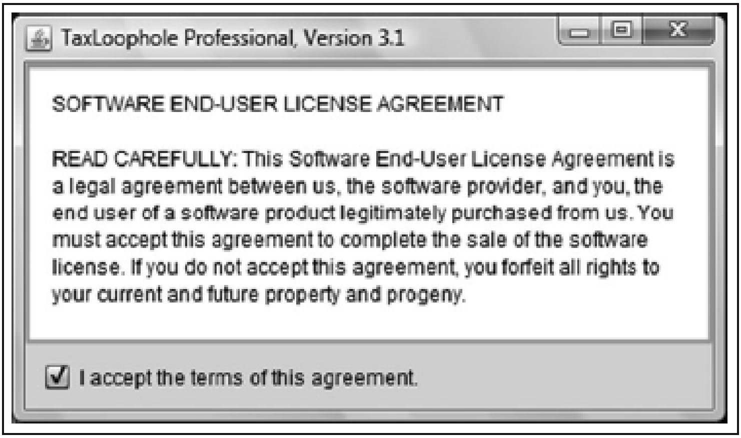
JTextArea component in Figure 18.13. Figure 18.14 contains the code associated with that component. Let’s now examine Figure 18.14’s code. Notice the \n\n in the JTextArea constructor call. As you might recall, \n’s are ignored inside JLabel text. But they work fine inside JTextArea text. Notice the three calls - setEditable(false), setLineWrap(true), and setWrapStyleWord(true). Those calls are common for JTextArea components.



```
private void createContents()
{
  JTextArea license;
  JCheckBox confirmBox;

  setLayout(new BorderLayout());
  license = new JTextArea(
    "SOFTWARE END-USER LICENSE AGREEMENT\n\n" +
    "READ CAREFULLY: This Software End-User License Agreement" +
    " is a legal agreement between us, the software provider," +
    " and you, the end user of a software product legitimately" +
    " purchased from us. You must accept this agreement to" +
    " complete the sale of the software license. If you do not" +
    " accept this agreement, you forfeit all rights to your" +
    " current and future property and progeny.");
  license.setEditable(false);
  license.setLineWrap(true);
  license.setWrapStyleWord(true);
  confirmBox = new JCheckBox(
    "I accept the terms of this agreement.", true);

  add(license, BorderLayout.CENTER);
  add(confirmBox, BorderLayout.SOUTH);
} // end createContents
```

Figure 18.14 The code that created Figure 18.13’s display

In Figure 18.13, notice how the blocks of text in the JTextArea and the JCheckBox components crowd up against their boundaries. The image in Figure 18.15 is 40 pixels higher and 20 pixels wider than the image in Figure 18.13. This enlargement enables the image in Figure 18.15 to include a 10-pixel wide empty border between each component's contents and its respective boundary.



SOFTWARE END-USER LICENSE AGREEMENT

READ CAREFULLY: This Software End-User License Agreement is a legal agreement between us, the software provider, and you, the end user of a software product legitimately purchased from us. You must accept this agreement to complete the sale of the software license. If you do not accept this agreement, you forfeit all rights to your current and future property and progeny.

☑ I accept the terms of this agreement.

Figure 18.15 Insertion of empty borders around contents of the components in Figure 18.13's window

The program containing the code for Figure 18.14 imports the javax.swing and java.awt packages. Suppose it also imports javax.swing.border.EmptyBorder. Then, the following statements can insert the space you see in Figure 18.15 between each component's boundary and its contents:

```
  license.setBorder(new
EmptyBorder(10, 10, 10, 10));
  confirmBox.setBorder(new
EmptyBorder(10, 10, 10, 10));
```

18.13 JCheckBox Component

User Interface

Look at the *check box* component at the bottom of Figure 18.15. Use a check box component if you want to present an option. A check box component displays a small square with a label to its right. When the square is blank, the check box is unselected. When the square contains a check mark, the check box is selected. Users click on the check box in order to toggle between selected and unselected.

Implementation

To create a check box component, call the JCheckBox constructor like this:

JCheckBox <*JCheckBox-reference*> = new JCheckBox(<*label*>, <*selected*>);

The *label* argument specifies the text that appears at the right of the check box's square. If the label argument is omitted, then no text appears at the right of the check box's square. The *selected* argument specifies whether the check box is selected initially—true means selected, false means unselected. If the selected argument is omitted, then the check box is initially unselected.

Here's how the check box was

created in the license-agreement window:

```
confirmBox = new JCheckBox("I accept the terms of this agreement.", true);
```

Methods

Here are the API headings and descriptions for the more popular JCheckBox methods:

`public boolean isSelected()`

> Returns true if the check box is selected and false otherwise.

`public void setVisible(boolean flag)`

> Makes the check box visible or invisible.

`public void setSelected(boolean flag)`

> Makes the check box selected or unselected.

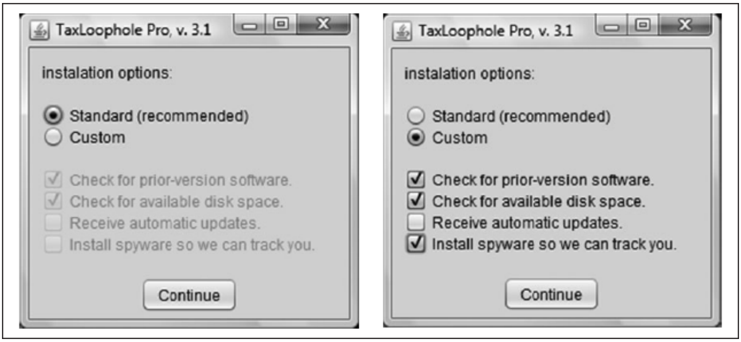
public void setEnabled(boolean flag)

Makes the check box enabled or disabled.

public void addActionListener(ActionListener listener)

Adds a listener to the check box.

The is Selected and setVisible methods are straightforward, but the other three methods need further explanation. Let's start with setSelected. Why might you want to call setSelected and adjust the selection status of a check box? Because you might want one user input to affect another user input. For example, in Figure 18.16, the user's selection of standard versus custom5 should affect the check box selections. More specifically, if the user selects the Standard option, the check box selections should go to their "standard" settings. As you can see in Figure 18.16's left window, the standard settings for the check boxes are the top two selected and the bottom two unselected. To have your program select the top two check boxes, those two check boxes should call setSelected(true). To have your program unselect the bottom two check boxes, those two check boxes should call setSelected(false).



instalation options:

● Standard (recommended)
○ Custom

☑ Check for prior-version software.
☑ Check for available disk space.
☐ Receive automatic updates.
☐ Install spyware so we can track you.

instalation options:

○ Standard (recommended)
● Custom

☑ Check for prior-version software.
☑ Check for available disk space.
☐ Receive automatic updates.
☑ Install spyware so we can track you.

Figure 18.16 Example that the illustrates JCheckBox's setSelected and setEnabled methods

To have your program disable a check box, the checkbox should call setEnabled(false). Why might you want to call setEnabled(false) and disable a check box? Because you might want to keep the user from modifying that box's value. For example, if the user selects the Standard option as shown in Figure 18.16's left window, the check box selections should be set to their standard settings (as explained above), and then each check box should call setEnabled(false). That way, the user cannot make changes to the standard-configuration check box values. In Figure 18.16's left window, note that the check boxes are gray. We say that they're *grayed out*. That's the standard GUI way of telling the user that something is disabled.

Check Box Listeners

With a JButton component, you'll almost always want an associated listener. But with a JCheckBox component, you may or may not want an associated listener. If you have a check box with no listener, then the check box simply serves as an input entity. If that's the case, then the check box's value (checked or unchecked) typically would get read and processed when the user clicks a button. On the other hand, if you want something to happen immediately, right when the user selects a check box, then add a listener to the check box component. Suppose you have a Green Background check box. If you want the window's background color to change to green right when the user clicks the check box, add a listener to the check box. The syntax for adding a listener to a JCheckBox component is the same as the syntax for adding a listener to a JButton component. Provide a listener that implements the ActionListener interface and then add the listener to the JCheckBox component by calling addActionListener.

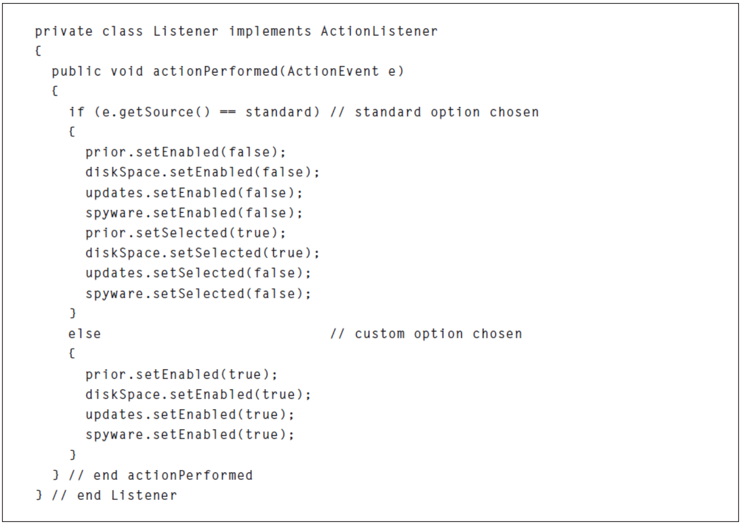
Be aware that the Java API provides an alternative listener interface for the

JCheckBox component—the ItemListener interface. An ActionListener listens for the user clicking on a check box. An ItemListener listens for a *state change*; that is, it listens for a check box changing from selected to unselected or vice versa. A check box state change is triggered when a user clicks the check box or when a program calls setSelected with a value that's different from the current value. Because we prefer the ActionListener interface in most situations, we'll stick with it when implementing JCheckBox listeners. When we get to the JRadioButton and JComboBox components in the next sections, we'll continue to use the ActionListener interface, not the ItemListener interface.

Installation-Options Example

It's now time to put these check box

concepts into practice by showing you some code. Look back at the installation-options windows in Figure 18.16. In Figure 18.17, we provide the listener code associated with those windows. Let's walk through the code. In the if statement's condition, we check to see whether the standard option was selected. If that's the case, we disable the check boxes by calling setEnabled(false) for each check box. We then assign the check boxes to their standard settings by calling setSelected(true) or setSelected(false) for each check box. In the else block, we handle the custom option being selected. We enable the check boxes by having each box call setEnabled(true). This enables the user to control whether the box is selected or not.



```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    if (e.getSource() == standard) // standard option chosen
    {
      prior.setEnabled(false);
      diskSpace.setEnabled(false);
      updates.setEnabled(false);
      spyware.setEnabled(false);
      prior.setSelected(true);
      diskSpace.setSelected(true);
      updates.setSelected(false);
      spyware.setSelected(false);
    }
    else                           // custom option chosen
    {
      prior.setEnabled(true);
      diskSpace.setEnabled(true);
      updates.setEnabled(true);
      spyware.setEnabled(true);
    }
  } // end actionPerformed
} // end Listener
```

Figure 18.17 Listener code for Figure 18.16’s installation-options windows

18.14 JRadioButton Component

User Interface

Look at the circles in the windows in Figure 18.16. They’re called *radio buttons*. A JRadioButton component displays a small circle with a label to its right. When the circle is blank, the radio button is unselected. When the circle contains a large dot, the radio button is selected.

According to the description so far, radio buttons sound a lot like check boxes. They display a shape and a label, and they keep track of whether something is on or off. The key difference between radio buttons and check boxes is that radio buttons almost always come in groups. And within a radio button group, only one radio button can be selected at a time. If a user clicks an unselected radio button, the clicked button becomes selected, and the previously selected button in the group becomes unselected. If a user clicks a selected radio button, no change occurs (i.e., the clicked button remains selected). In contrast, if a user clicks a selected check box, the check box changes its state from selected to unselected.

Implementation

To create a JRadioButton component, call the JRadioButton constructor like this:

```
JRadioButton <JRadioButton-reference> =
  new JRadioButton(<label>, <selected>);
```

The *label* argument specifies the text that appears at the right of the radio button's circle. If the label argument is omitted, then no text appears at the right of the radio button's circle. The *selected* argument specifies whether the radio button is initially selected—true means selected, false means unselected. If the selected argument is omitted, then the radio button is initially unselected.

This example shows how we created the standard and custom radio buttons in the installation-options program:

```
standard = new JRadioButton("Standard
(recommended)", true);
custom = new JRadioButton("Custom");
```

To enable the functionality of a radio button group (that is, only one button selected at a time), create a ButtonGroup object and add individual radio button components to it. Here's how:

```
ButtonGroup <ButtonGroup-reference> = new ButtonGroup();
<ButtonGroup-reference>.add(<first-button-in-group>);
...
<ButtonGroup-reference>.add(<last-button-in-group>);
```

The following example shows how we created the radio button group for the standard and custom radio buttons in the installation-options program:

```
ButtonGroup rbGroup = new ButtonGroup();
```

```
rbGroup.add(standard);
rbGroup.add(custom);
```

After adding radio buttons to a radio button group, you still have to add them to a container. Radio buttons work the same as other components in terms of adding them to a container. Call the container's add method like this:

```
add(<first-button-in-group>);
...
add(<last-button-in-group>);
```

⚠ That's a lot of adding. You need to add each radio button twice—once to a radio button group and once to a container. If you like shortcuts, you might be thinking, Why does Java make you add the individual radio buttons to the container? Why not add the button group to the container and rely on that to take care of getting the buttons in the container? Adding the buttons individually to the container gives you freedom in positioning the buttons. If you wanted to, you could even put them in different panels.

Because the JRadioButton class begins with a *J,* you can correctly assume that it's defined in the javax.swing package. But what about the ButtonGroup class? Even though it doesn't begin with a *J,* it's also defined in the javax.swing package.

Methods

Here are the API headings and descriptions for the more popular JRadioButton methods:

public boolean isSelected()

> Returns true if the radio button is selected and false otherwise.

public void setSelected(boolean flag)

> Makes the radio button selected if the argument is

true. Does nothing if the argument is false.

public void setEnabled(boolean flag)

Makes the radio button enabled or disabled. If enabled, it responds to mouse clicks.

public void addActionListener(ActionListener listener)

Adds a listener to the radio button.

We described these same methods in the JCheckBox section. Only one of them needs further attention—the setSelected method. To understand how setSelected works, you first need to understand fully how a user interacts with a radio button group. To select a radio button, a user clicks it. That causes the radio button to become selected and all other radio buttons in the group to become unselected. To programmatically select a radio button, you have the radio button call setSelected(true). That causes the radio button to become selected and all other radio buttons in the group to become unselected. As mentioned above, there is no way for a user to unselect a button. Likewise, there is no way for a program to unselect a button. That's why calling setSelected(false) doesn't do anything. It compiles and runs, but it doesn't cause any buttons to change their selected status.

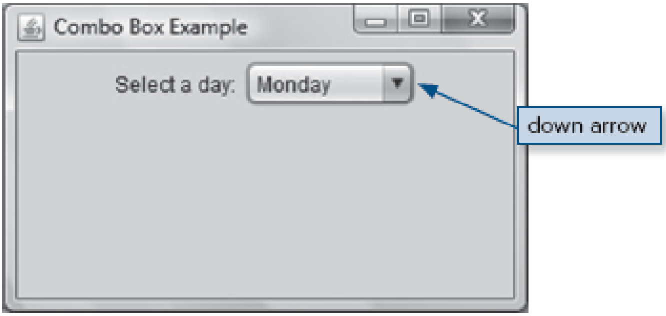
18.15 JComboBox Component

User Interface

A *combo box* allows a user to select an item from a list of items. Combo boxes are sometimes called *drop-down lists* because if a user clicks a combo box's down arrow, a list of selection items drops down from the original display. Then, if a user clicks a selection from the drop-down list, the list disappears, and only the selected item remains displayed. To get a better idea of what we're talking about, see the select-a-day combo box in Figure 18.18.

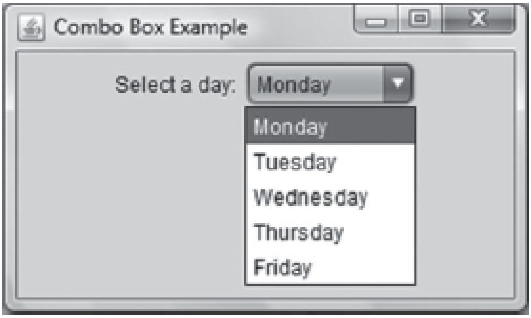
Combo boxes and radio button groups are similar in that they both allow the user to select one item from a list of items. But a combo box takes up less space on the window. So if you have a long list of items to choose from, and you want to save space, use a combo box rather than a group of radio buttons.

Initial display:


Combo Box Example
down arrow

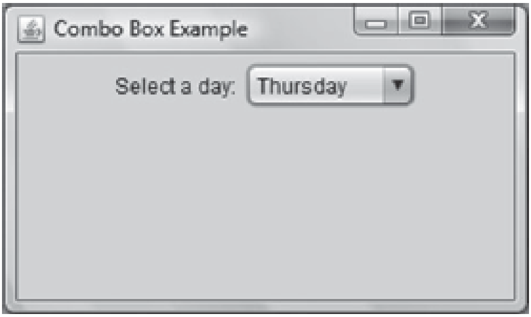
Select a day: Monday

After a user clicks the down arrow:


Combo Box Example
Monday
Tuesday
Wednesday
Thursday
Friday

Select a day: Monday

After a user clicks Thursday:


Combo Box Example

Select a day: Thursday

Figure 18.18 Select-a-day combo box example

Implementation

Creating a combo box component is a two-step process. First, instantiate an array of options. Then, use the array as part of a JComboBox instantiation. Here’s the syntax for a typical JComboBox instantiation:

JComboBox <JComboBox-reference> = new JComboBox(<array-of-options>);

The following code fragment shows how we created the combo box in Figure 18.18:

```
String[] days =
  {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
daysBox = new JComboBox(days);
```

When a combo box first displays, the first item in its array is selected. So in the above example, Monday is selected when the combo box first displays.

Methods

Here are the API headings and descriptions for the more popular

JComboBox methods:

public void setVisible(boolean flag)

Makes the combo box visible or invisible.

public void setEditable(boolean flag)

Makes the combo box's top portion editable or non-editable.

public Object getSelectedItem()

Returns the item that is currently selected.

public void setSelectedItem(Object item)

Changes the currently selected item to the item that's passed in.

public int getSelectedIndex()

Returns the index of the item that is currently selected.

public void setSelectedIndex(int index)

Changes the currently selected item to the item at the given index position.

public void addActionListener(ActionListener listener)

Adds a listener to the combo box.

The setVisible and addActionListener methods should look familiar by now. The other methods are new and require further explanation. Let's start with setEditable. If a combo box calls setEditable(true), the combo box's top portion becomes editable. That means that a user can enter text into it the same as if it were a text box component. Additionally, the user still can use the drop-down portion of the combo box the same as always. If the user enters text into the top portion of an editable combo box, the drop-down options don't change, but the selected value does change to the entered value, as evidenced by the returned value from getSelectedItem. Combo boxes are named “combo” for “combination” because they are capable of implementing a mixture of components—part drop-down list, part text box. But most programmers don’t bother with the combo box’s text-box capability. They usually stick with the default behavior, where the top portion of the combo box is not editable.

The getSelectedItem method returns the currently selected item. For example, here’s how you can retrieve the currently selected daysBox item and store it in a favoriteDay variable:

```
String favoriteDay = (String) daysBox.getSelectedItem();
```

What’s the point of the (String) cast operator? The getSelectedItem method is defined to have a return ⚠ type of

Object. Therefore, if there were no cast operator, the compiler would see an Object at the right being assigned into a String at the left, and that would generate a compile-time error. But there is a cast operator, so the compiler sees a String at the right being assigned into a String at the left. And that makes the compiler happy.

If you'd like to select an option from a combo box programmatically, call setSelectedItem and pass in the item that you want to select. For example, to select Friday from the daysBox component, do this:

```
daysBox.setSelectedItem("Friday");
```

Normally, you'll call setSelectedItem with an argument that matches one of the combo box's items. But that's not always the case. If you'd like to clear a combo box so that no options are selected, call setSelectedItem(null). If you call setSelectedItem with a different item (not null and not a combo box item), then nothing happens. Well, actually, nothing happens if it's a standard combo box. But if you first make it an editable combo box by calling setEditable(true), then the item you pass to setSelectedItem gets put into the editable top portion of the combo box.

There are two ways to access items in a combo box. Use either item names or item indexes. The getSelectedItem and setSelectedItem methods use item names. The getSelectedIndex and setSelectedIndex methods use item indexes. For example, note how this code fragment calls setSelectedIndex with an index value of 2:

```
String[] days =
  {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
daysBox = new JComboBox(days);
daysBox.setSelectedIndex(2);
```

Because combo boxes store their items in arrays, combo box item indexes are 0-based. Therefore, in this code fragment, Monday is 0, Tuesday is 1, and Wednesday is 2. Thus, daysBox.setSelectedIndex(2) changes the selected item to Wednesday.

Indexing helps you process data.

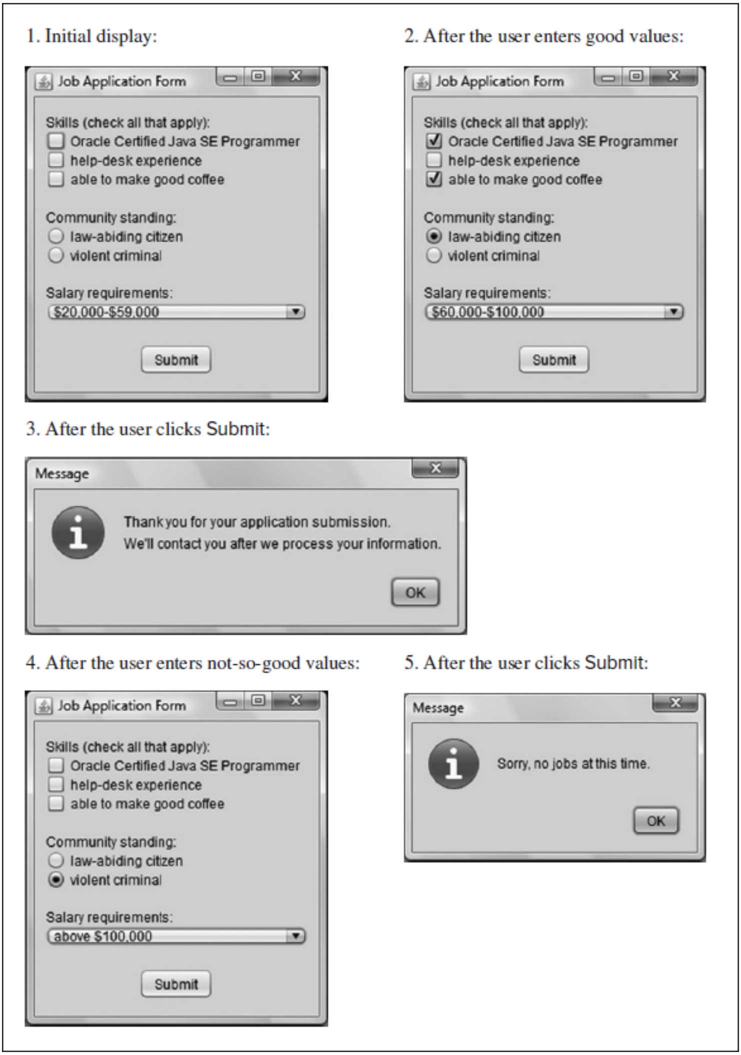
Now for a short brain-teaser. Given the above code fragment, how can you change the currently selected day to the next day? Do arithmetic with the index, like this:

```
daysBox.setSelectedIndex(daysBox.getSele
+ 1);
```

18.16 Job Application Example

In this section, we put into practice what you've learned in the previous three sections. We present a complete program that uses check boxes, radio buttons, and a combo box. The program implements a job application form. If the user enters values that are indicative of a good employee, the program displays an encouraging message (“Thank you for your application submission. We’ll contact you after we process your information.”). Study the sample session in Figure 18.19 to get a better idea of what we’re talking about.

See the JobApplication program listing in Figures 18.20a, 18.20b, and 18.20c. You should peruse the entire program on your own, particularly the listener code, but we’ll focus only on the most difficult part—the layout design.



1. Initial display:

Job Application Form

Skills (check all that apply):
☐ Oracle Certified Java SE Programmer
☐ help-desk experience
☐ able to make good coffee

Community standing:
○ law-abiding citizen
○ violent criminal

Salary requirements:
$20,000-$59,000

2. After the user enters good values:

Job Application Form

Skills (check all that apply):
☑ Oracle Certified Java SE Programmer
☐ help-desk experience
☑ able to make good coffee

Community standing:
● law-abiding citizen
○ violent criminal

Salary requirements:
$60,000-$100,000

3. After the user clicks Submit:

Message

Thank you for your application submission.
We'll contact you after we process your information.

4. After the user enters not-so-good values:

Job Application Form

Skills (check all that apply):
☐ Oracle Certified Java SE Programmer
☐ help-desk experience
☐ able to make good coffee

Community standing:
○ law-abiding citizen
● violent criminal

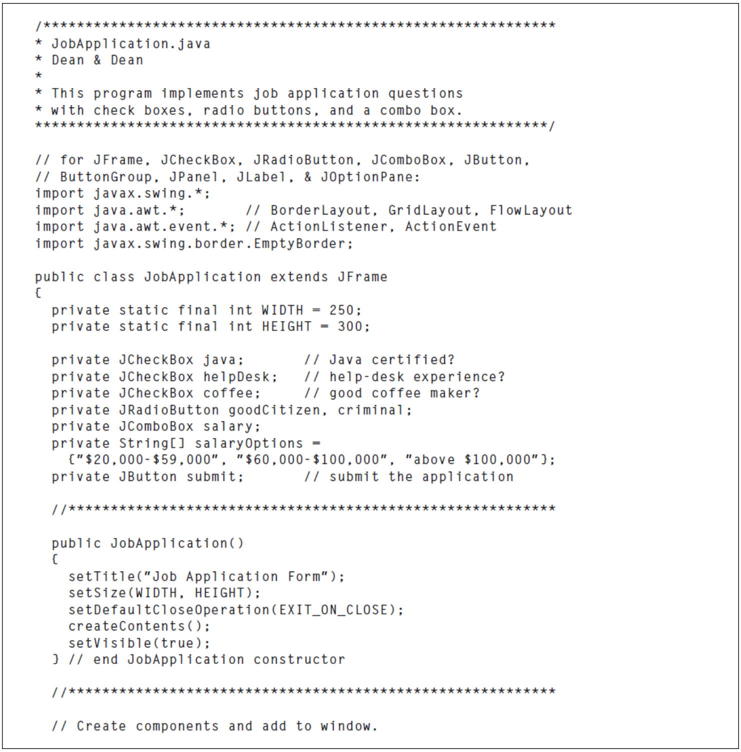
Salary requirements:
above $100,000

5. After the user clicks Submit:

Message

Sorry, no jobs at this time.

Figure 18.19 Sample session for the JobApplication program



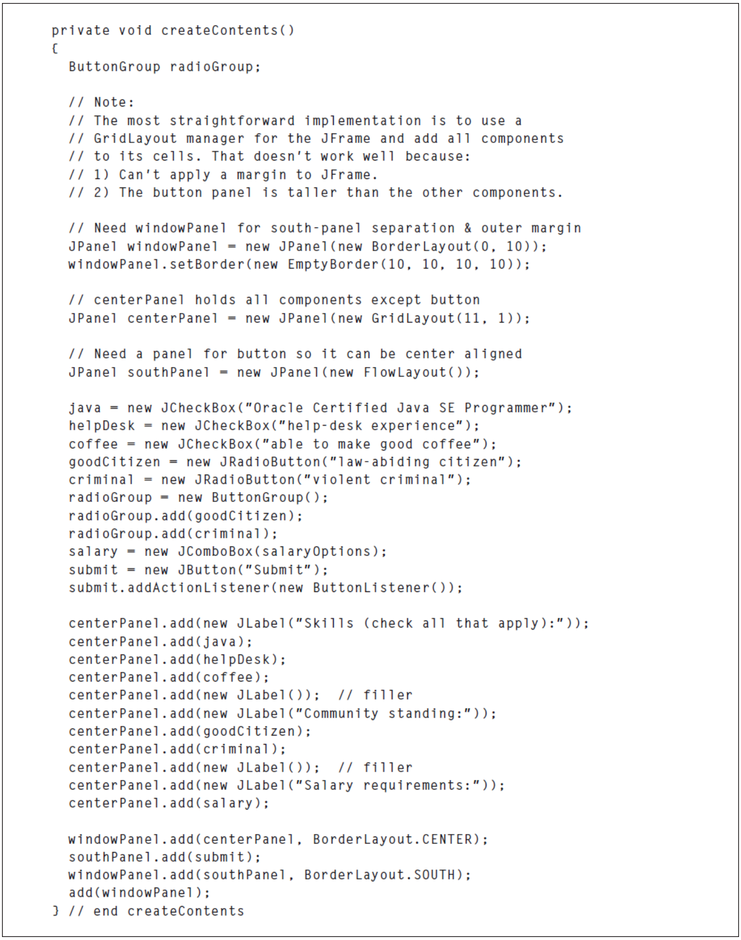
```
/*************************************************************
* JobApplication.java
* Dean & Dean
*
* This program implements job application questions
* with check boxes, radio buttons, and a combo box.
*************************************************************/

// for JFrame, JCheckBox, JRadioButton, JComboBox, JButton,
// ButtonGroup, JPanel, JLabel, & JOptionPane:
import javax.swing.*;
import java.awt.*;       // BorderLayout, GridLayout, FlowLayout
import java.awt.event.*; // ActionListener, ActionEvent
import javax.swing.border.EmptyBorder;

public class JobApplication extends JFrame
{
  private static final int WIDTH = 250;
  private static final int HEIGHT = 300;

  private JCheckBox java;       // Java certified?
  private JCheckBox helpDesk;   // help-desk experience?
  private JCheckBox coffee;     // good coffee maker?
  private JRadioButton goodCitizen, criminal;
  private JComboBox salary;
  private String[] salaryOptions =
    {”$20,000-$59,000”, ”$60,000-$100,000”, ”above $100,000”};
  private JButton submit;       // submit the application

  //***********************************************************

  public JobApplication()
  {
    setTitle(”Job Application Form”);
    setSize(WIDTH, HEIGHT);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end JobApplication constructor

  //***********************************************************

  // Create components and add to window.
```

Figure 18.20a JobApplication program—part A

If at first you don't succeed, try again.

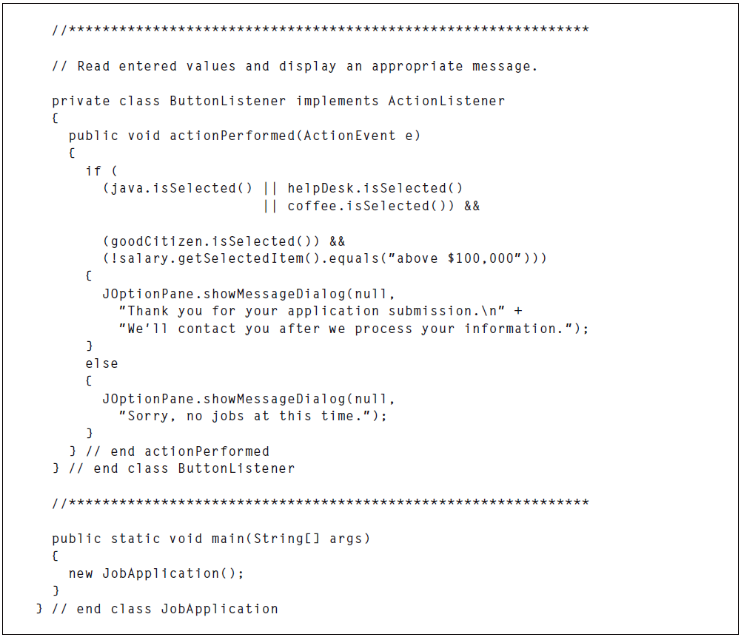
We spent quite a bit of time on the JobApplication's layout in order to get things to look right. Initially, we thought a simple one-column GridLayout manager would work. We added one component per cell, and we added three filler components (empty

JLabels) to create gaps between the four different input areas. We thought that plan would yield the layout shown in Figure 18.21’s left picture. Unfortunately, when we entered the code, the actual program yielded the layout shown in Figure 18.21’s right picture. There are three problems with the actual layout—the Submit button is too wide, the top two gaps are missing, and the components are touching the top and left boundaries. We’ll now discuss how to fix those problems.



```
private void createContents()
{
  ButtonGroup radioGroup;

  // Note:
  // The most straightforward implementation is to use a
  // GridLayout manager for the JFrame and add all components
  // to its cells. That doesn’t work well because:
  // 1) Can’t apply a margin to JFrame.
  // 2) The button panel is taller than the other components.

  // Need windowPanel for south-panel separation & outer margin
  JPanel windowPanel = new JPanel(new BorderLayout(0, 10));
  windowPanel.setBorder(new EmptyBorder(10, 10, 10, 10));

  // centerPanel holds all components except button
  JPanel centerPanel = new JPanel(new GridLayout(11, 1));

  // Need a panel for button so it can be center aligned
  JPanel southPanel = new JPanel(new FlowLayout());

  java = new JCheckBox(”Oracle Certified Java SE Programmer”);
  helpDesk = new JCheckBox(”help-desk experience”);
  coffee = new JCheckBox(”able to make good coffee”);
  goodCitizen = new JRadioButton(”law-abiding citizen”);
  criminal = new JRadioButton(”violent criminal”);
  radioGroup = new ButtonGroup();
  radioGroup.add(goodCitizen);
  radioGroup.add(criminal);
  salary = new JComboBox(salaryOptions);
  submit = new JButton(”Submit”);
  submit.addActionListener(new ButtonListener());

  centerPanel.add(new JLabel(”Skills (check all that apply):”));
  centerPanel.add(java);
  centerPanel.add(helpDesk);
  centerPanel.add(coffee);
  centerPanel.add(new JLabel());  // filler
  centerPanel.add(new JLabel(”Community standing:”));
  centerPanel.add(goodCitizen);
  centerPanel.add(criminal);
  centerPanel.add(new JLabel());  // filler
  centerPanel.add(new JLabel(”Salary requirements:”));
  centerPanel.add(salary);

  windowPanel.add(centerPanel, BorderLayout.CENTER);
  southPanel.add(submit);
  windowPanel.add(southPanel, BorderLayout.SOUTH);
  add(windowPanel);
} // end createContents
```

Figure 18.20b JobApplication program—part B



```
  //**************************************************************

  // Read entered values and display an appropriate message.

  private class ButtonListener implements ActionListener
  {
    public void actionPerformed(ActionEvent e)
    {
      if (
        (java.isSelected() || helpDesk.isSelected()
                           || coffee.isSelected()) &&

        (goodCitizen.isSelected()) &&
        (!salary.getSelectedItem().equals("above $100,000")))
      {
        JOptionPane.showMessageDialog(null,
          "Thank you for your application submission.\n" +
          "We'll contact you after we process your information.");
      }
      else
      {
        JOptionPane.showMessageDialog(null,
          "Sorry, no jobs at this time.");
      }
    } // end actionPerformed
  } // end class ButtonListener

  //**************************************************************

  public static void main(String[] args)
  {
    new JobApplication();
  }
} // end class JobApplication
```

Figure 18.20c JobApplication program—part C

Problem 1: Submit Button is Too Wide

Embed another manager.

As you may recall from earlier in the chapter, buttons expand if they're added directly to a GridLayout cell. That explains the wide Submit button. You can fix this cosmetic problem by embedding a FlowLayout panel into the Submit button's area, and then adding the Submit button to the FlowLayout panel. With a FlowLayout manager, buttons don't expand; they keep their natural size.

Problem 2: Top Two Gaps are Missing

In our first cut of the program, we used this code to implement the filler components:

```
JLabel filler = new JLabel();
...
add(filler);
...
add(filler);
...
add(filler);
```

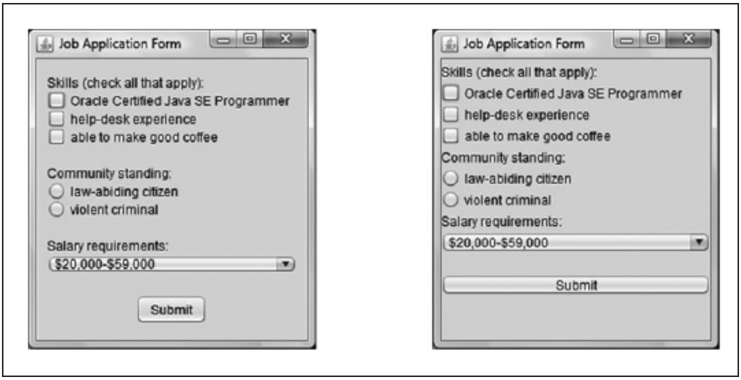
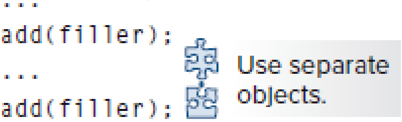

Job Application Form
Skills (check all that apply):
☐ Oracle Certified Java SE Programmer
☐ help-desk experience
☐ able to make good coffee
Community standing:
○ law-abiding citizen
○ violent criminal
Salary requirements:
$20,000-$59,000
Submit
Job Application Form
Skills (check all that apply):
☐ Oracle Certified Java SE Programmer
☐ help-desk experience
☐ able to make good coffee
Community standing:
○ law-abiding citizen
○ violent criminal
Salary requirements:
$20,000-$59,000
Submit

Figure 18.21 Intended versus actual layouts with the 13-row by 1-column GridLayout scheme



```
...
add(filler);
...
add(filler);
```

Use separate objects.

We instantiated only one label and reused it three times. You like to reuse, right? Well, the layout manager doesn't. The layout manager sees only one object, so it makes only one cell. It does not make cells for the first two add(filler) calls; it only makes a cell for the last add(filler) call. You can fix the problem by using three anonymous JLabel objects like this:

```
add(new JLabel());
...
```

```
add(new JLabel());
...
add(new JLabel());
```

Problem 3: Components Are Too Close to the Top and Left Boundaries

By default, containers have no margins. So if a container has left-aligned components, those components will be very close to the container's boundaries. That explains the boundary ugliness in Figure 18.21's right picture. Just as we did for Figure 18.15, you can add a margin by calling setBorder like this:

<*container*>.setBorder(new EmptyBorder(<*top*>, <*left*>, <*bottom*>, <*right*>));

In calling setBorder, you'll need to pass a border object as an argument. There are several different types of border classes. You should use the

EmptyBorder class because an empty border produces a margin, which is what you want. In calling the EmptyBorder constructor, you'll need to pass in pixel values for the widths of the border's top, left, bottom, and right sides. For example, this constructor call passes in 10-pixel values for all four of the border's sides:

```
windowPanel.setBorder(new
EmptyBorder(10, 10, 10, 10));
```

Be aware that the EmptyBorder class is in the javax.swing.border package. So import that package if you want to create an empty border.

Use a panel.

You might think that the setBorder method works for all containers. Not so. It works for the JPanel container, but not the JFrame container. Therefore, you need to add a JPanel container to the JobApplication JFrame window and call setBorder from the JPanel container.

What type of layout manager is appropriate for the new JPanel container? If you use a GridLayout manager, that works OK, but not great. With a GridLayout, all rows are the same height. In Figure 18.19, note how the Submit button is slightly taller than the other components. The Submit button's added height provides a visual cue for the button's importance. To accommodate the button being taller than the other components, use a BorderLayout manager. Add the button panel to the south region and add all the other components to the center region. Actually, because the center region allows for only one component, you need to add the components to a GridLayout panel and then add the GridLayout panel to the center region.

Document Difficult Code

The JobApplication program's layout

design is rather complicated and somewhat non-intuitive. If you ever write complicated and non-intuitive code, you should document it with detailed comments. If you don’t, then someone (maybe you) might waste time later in trying to figure it out. See all the comments for the panel declarations in Figure 18.20b. Those comments help to clarify the layout-design code.

18.17 More Swing Components

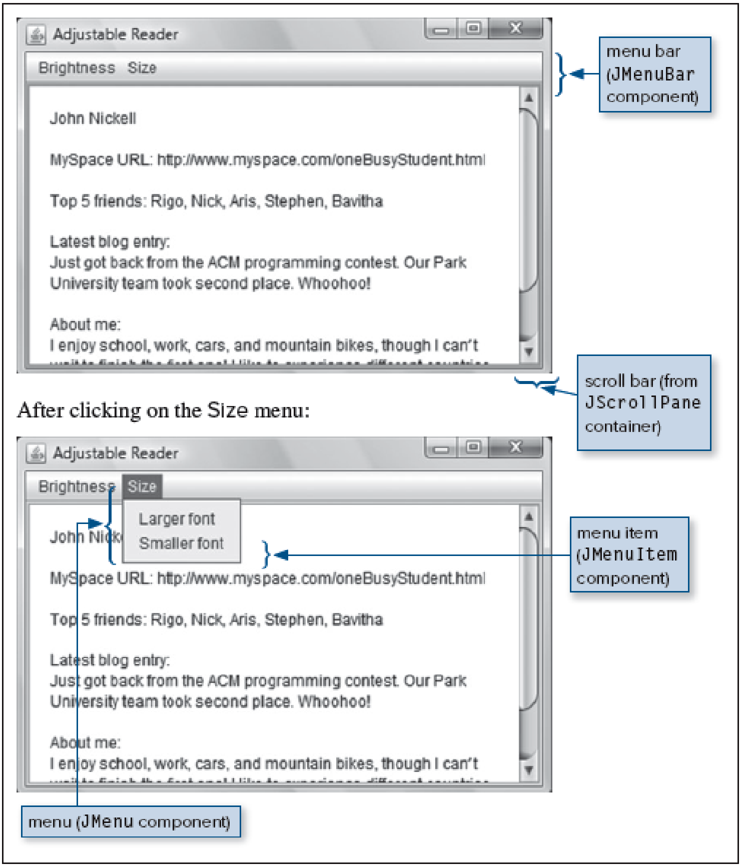
In this chapter and the previous chapter, you’ve learned quite a bit about the Swing library. Enough to get up and running for most basic GUI needs. If you decide you want to know more, refer to Oracle’s Java API website. In particular, refer to this web page within Oracle’s Java API website:

http://download.oracle.com/javase/tutorial/uiswing/components/componentlist.html

It contains links to web pages for each of the standard Swing components. Each Swing component web page provides detailed information about the component, with picture examples. By perusing the Swing components web pages now, you'll know what's available.

Menus and Scroll Panes

As a first attempt at learning Swing components on your own, we recommend that you look up the -JMenuBar, JMenu, and JMenuItem classes on the Java API website. Those classes allow you to add a *menu bar* and *menus* to the top of a window. Also, look up the JScrollPane class. It allows you to create a scrollable container. See Figure 18.22. It shows a window with a menu bar and a scroll bar. The menu bar contains two menus—one allows the user to adjust the brightness of the window’s background color and one allows the user to adjust the font size of the window’s text. The scroll bar is part of what is known as a *scroll pane*. The scroll bar allows the user to scroll up and down and view the contents of the entire window.


Adjustable Reader
Brightness
Size
menu bar (JMenuBar component)
scroll bar (from JScrollPane container)

John Nickell

MySpace URL: http://www.myspace.com/oneBusyStudent.html

Top 5 friends: Rigo, Nick, Aris, Stephen, Bavitha

Latest blog entry:
Just got back from the ACM programming contest. Our Park University team took second place. Whoohoo!

About me:
I enjoy school, work, cars, and mountain bikes, though I can't

After clicking on the Size menu:

Adjustable Reader
Brightness
Size
Larger font
Smaller font
menu item (JMenuItem component)
menu (JMenu component)

John Nickell

MySpace URL: http://www.myspace.com/oneBusyStudent.html

Top 5 friends: Rigo, Nick, Aris, Stephen, Bavitha

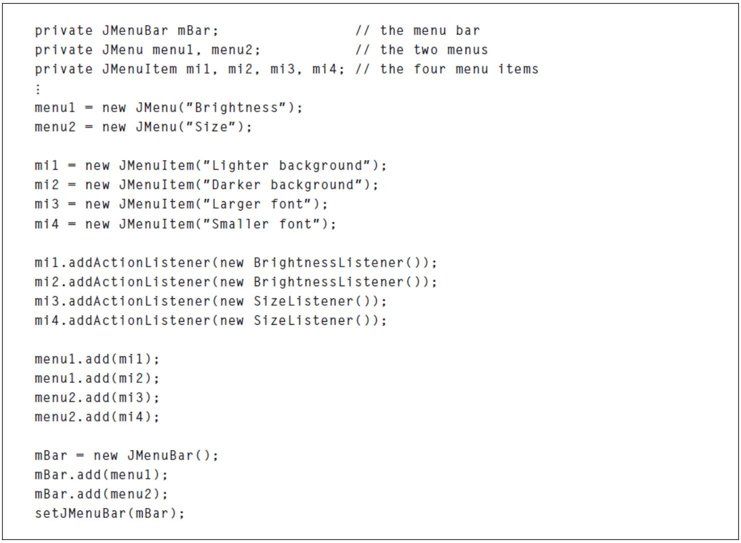
Latest blog entry:
Just got back from the ACM programming contest. Our Park University team took second place. Whoohoo!

About me:
I enjoy school, work, cars, and mountain bikes, though I can't

Figure 18.22 A reader program window that uses a menu bar and a scroll bar to adjust the view

If you'd like to see Figure 18.22's program in its entirety, view the ReaderMenu.java file on the book's website. Figure 18.23 shows a portion of that program—the portion that creates the menu bar, menus, and menu items. And the following statement shows the portion of the program that creates the scroll pane. More specifically, the following statement creates a scroll pane for a text area component and then adds the scroll pane to the window.

add(new JScrollPane(textArea));



```
private JMenuBar mBar;              // the menu bar
private JMenu menu1, menu2;         // the two menus
private JMenuItem mi1, mi2, mi3, mi4; // the four menu items
:
menu1 = new JMenu("Brightness");
menu2 = new JMenu("Size");

mi1 = new JMenuItem("Lighter background");
mi2 = new JMenuItem("Darker background");
mi3 = new JMenuItem("Larger font");
mi4 = new JMenuItem("Smaller font");

mi1.addActionListener(new BrightnessListener());
mi2.addActionListener(new BrightnessListener());
mi3.addActionListener(new SizeListener());
mi4.addActionListener(new SizeListener());

menu1.add(mi1);
menu1.add(mi2);
menu2.add(mi3);
menu2.add(mi4);

mBar = new JMenuBar();
mBar.add(menu1);
mBar.add(menu2);
setJMenuBar(mBar);
```

Figure 18.23 Code that creates menu bar, menus, and menu items for the window in Figure 18.22

Sliders

For another learn-on-your-own

example, we recommend that you look up the JSlider class on the Java API website. The JSlider class allows you to add a *slider* component to a window. A slider allows the user to select a value from a range of values. To select a value, the user drags a “thumb” along a bar of values. See Figure 18.24. It mimics a lunar eclipse by covering a white circle (the moon) with a gray circle (the Earth’s shadow). When the user drags the slider’s thumb right, the shadow moves right. When the user drags the slider’s thumb left, the shadow moves left. The slider uses an event handler to adjust the shadow’s position.

If you’d like to see Figure 18.24’s program in its entirety, view the EclipseSlider.java file on the book’s website. The following code from the lunar eclipse program shows how to instantiate a slider, set properties, and add a listener:

```
slider = new
JSlider(SwingConstants.HORIZONTAL,
0, 100, 0);
   slider.setMajorTickSpacing(5);
   slider.setPaintTicks(true);
   slider.addChangeListener(new
Listener());
```

This adds the slider to the current JFrame:

```
   add(slider, BorderLayout.SOUTH);
```

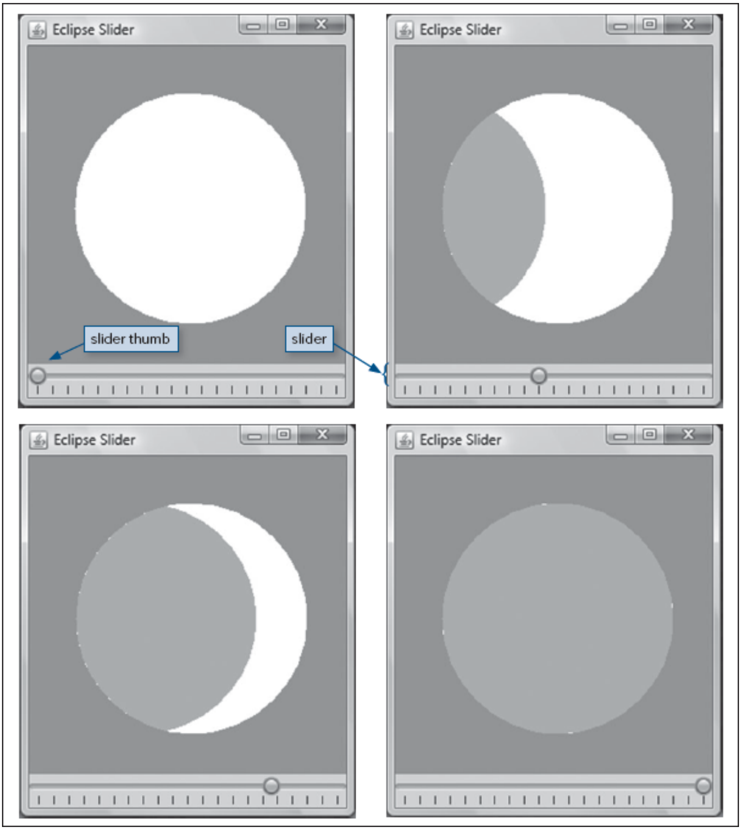

Eclipse Slider
slider thumb
slider
Eclipse Slider
Eclipse Slider
Eclipse Slider

Figure 18.24 Four displays of a window that uses a slider to mimic an eclipse

Summary

- Layout managers automate the positioning of components within containers.
- The FlowLayout class

implements a simple one-compartment layout scheme that allows multiple components to be appended to the right end of a row that wraps onto subsequent rows.

- The BoxLayout manager implements a simple one-row or one-column layout scheme.
- The BorderLayout manager provides five regions/compartments—north, south, east, west, and center—in which to insert components.
- The SwingConstants interface stores a set of GUI-related constants that are commonly used by many different GUI programs.
- The GridLayout manager lays out a container's components in a rectangular grid of equal-sized cells. Each cell can hold only one component.
- If you have a complicated

window with many components, you might want to compartmentalize them by storing groups of components in JPanel containers.

- To display multiple lines of text, use a JTextArea component.
- A JCheckBox component displays a small square with an identifying label. Users click the check box in order to toggle it between selected and unselected.
- A JRadioButton component displays a small circle with a label to its right. If an unselected button is clicked, the clicked button becomes selected, and the previously selected button in the group becomes unselected.
- A JComboBox component allows the user to select an item from a list of items. JComboBox components are called “combo boxes” because they are a

combination of a text box (normally, they look just like a text box) and a list (when the down arrow is clicked, they look like a list).

Review Questions

---

§18.2 GUI Design and Layout Managers

1. Layout managers adapt automatically to changes in the size of a container or one of its components. (T / F)

2. Which package contains layout managers?

§18.3 FlowLayout Manager

3. How does the FlowLayout manager arrange components?

4. Write a single statement that gives the current container a flow layout with right alignment.

§18.4 BoxLayout Manager

5. Write a statement that gives a Frame object called frame a BoxLayout manager that aligns elements in a column.

§18.5 BorderLayout Manager

6. What are the five regions established by the BorderLayout manager?

7. The sizes of the five regions in a border layout are determined at runtime based on the contents of the four outer regions. (T / F)

8. By default, how many components can you put in any one region of a border layout?

9. Write a single statement that adds a new JLabel with the text “Stop” to the center region of a BorderLayout manager. The label should be centered within the center region.

§18.6 GridLayout Manager

10. When you instantiate a GridLayout manager, you should always specify both the number of rows and the number of columns. (T / F)

11. In a grid layout, all cells are the same size. (T / F)

§18.7 Tic-Tac-Toe Example

12. What happens to the xTurn variable in the Tic-Tac-Toe program if you click the same cell twice?

§18.10 JPanel Class

13. Why are JPanel containers particularly useful with GridLayout and BorderLayout windows (as opposed to FlowLayout windows)?

§18.11 MathCalculator Program

14. In the MathCalculator program's createContents method, indicate the purpose of the statement: add(new JLabel());

§18.12 JTextArea Component

15. JTextArea components are editable by default. (T / F).

16. JTextArea components employ line wrap by default. (T / F).

§18.13 JCheckBox Component

17. What happens if you click a check box that's already selected?

18. Provide a statement that creates a check box named attendance. The check box should be preselected, and it should have an "I will attend" label.

§18.14 JRadioButton Component

19. What happens if you click a radio button that is already selected?

20. What happens if you click an initially unselected radio button that is a member of a RadioGroup?

§18.15 JComboBox Component

21. How are combo boxes and radio button groups similar?

22. What two methods can be called to determine the current selection for a combo box?

§18.16 Job Application Example

23. The JobApplication program contains the following code fragment. What happens to the program if the code fragment is omitted?

```
radioGroup = new ButtonGroup();
```

```
radioGroup.add(goodCitizen);
radioGroup.add(criminal);
```

24. Provide a statement that adds a 20-pixel blank margin to a JPanel container named panel.

§18.17 More Swing Components

25. Provide a JSlider constructor call where the minimum value is 0, the maximum value is 50, and the initial value is 10. Hint: Look up the answer on Oracle's Java API website.

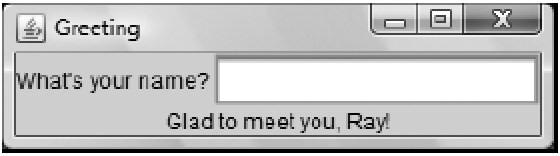
Exercises

---

1. [after §18.2] What is the default layout manager for a JFrame window?

2. [after §18.3] With a FlowLayout manager, a button component expands so that it completely fills the size of the region in which it is placed. (T / F)

3. [after §18.5] Provide a complete program that is a modification of Chapter 17’s Greeting program. Your new program should use a BorderLayout manager (instead of a FlowLayout manager), and it should generate the following display after a name has been entered. Make the frame size 300 pixels wide and 80 pixels high.



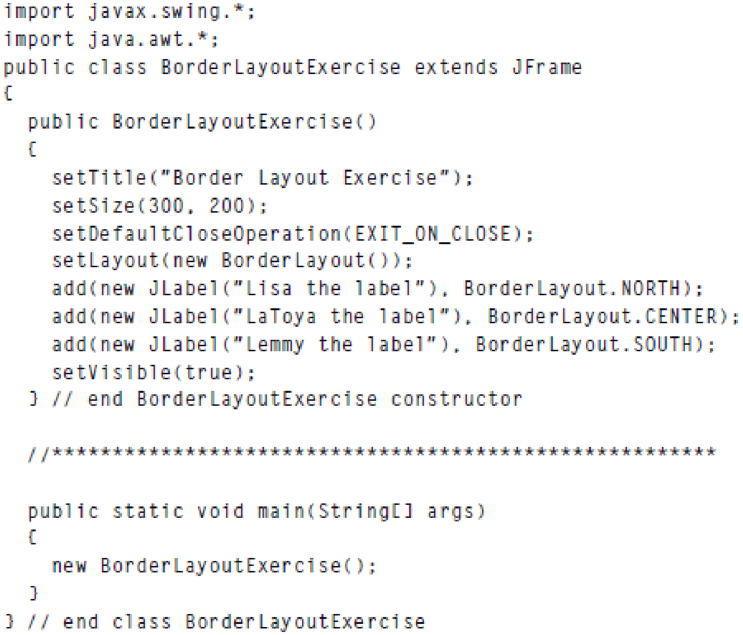
What's your name?

Glad to meet you, Ray!

4. [after §18.5] With a BorderLayout, what happens if the east region is empty? Said another way, which region(s), if any, expand(s) if the east region is empty?

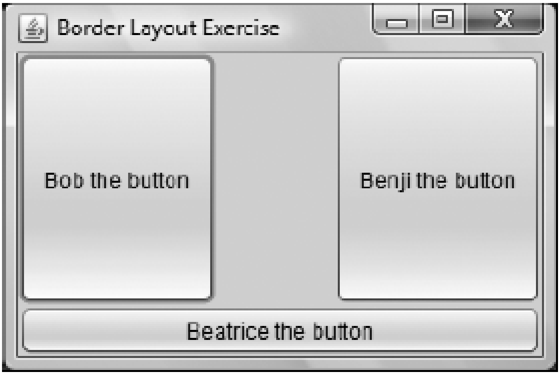
5. [after §18.5] Assume you have

this program:

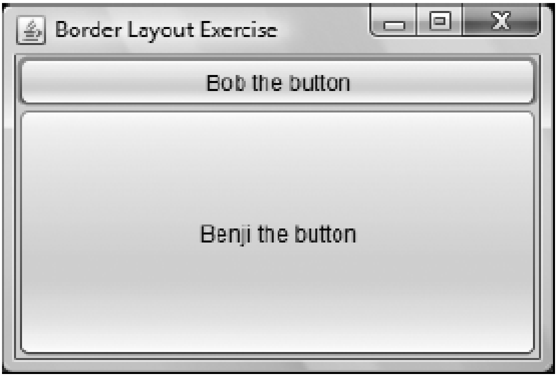


```
import javax.swing.*;
import java.awt.*;
public class BorderLayoutExercise extends JFrame
{
  public BorderLayoutExercise()
  {
    setTitle("Border Layout Exercise");
    setSize(300, 200);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setLayout(new BorderLayout());
    add(new JLabel("Lisa the label"), BorderLayout.NORTH);
    add(new JLabel("LaToya the label"), BorderLayout.CENTER);
    add(new JLabel("Lemmy the label"), BorderLayout.SOUTH);
    setVisible(true);
  } // end BorderLayoutExercise constructor

  //*******************************************************

  public static void main(String[] args)
  {
    new BorderLayoutExercise();
  }
} // end class BorderLayoutExercise
```

a) Specify the changes you would make to the above code to produce this output:



b) Specify the changes you would make to the above code to produce this output:


Border Layout Exercise
Bob the button
Benji the button

6. [after §18.6] If a JButton component is directly added to a GridLayout cell, it expands so that it completely fills the size of its cell. (T / F)

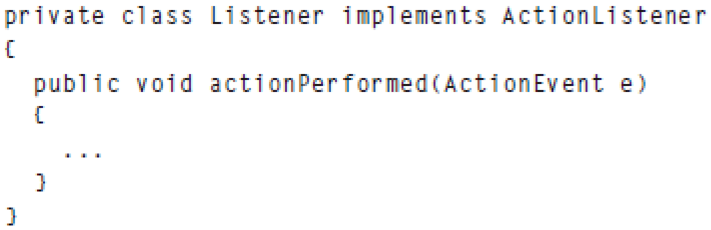
7. [after §18.6] Given the following code fragment, draw a picture that illustrates the buttons' positions within the program's window.

```
setLayout(new GridLayout(0, 3));
add(new JButton("1"));
add(new JButton("2"));
add(new JButton("3"));
```

```
  add(new JButton("4"));
  add(new JButton("5"));
  add(new JButton("6"));
  add(new JButton("7"));
```

8. [after §18.10] What kind of container should you put into an individual grid layout cell or an individual border layout region to allow that cell or region to contain more than one component?

9. [after §18.12] Suppose you're given a window with two JTextArea components, msg1 and msg2, and a JButton component. When clicked, the button swaps the contents of the two text areas. Provide the code that performs the swap operation. More specifically, provide the code that goes inside the following actionPerformed method:

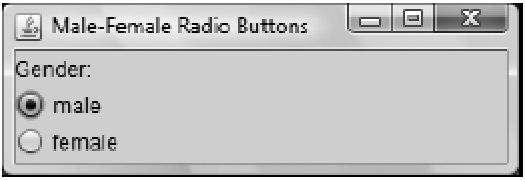


```
private class Listener implements ActionListener
{
  public void actionPerformed(ActionEvent e)
  {
    ...
  }
}
```

10. [after §18.13] Provide a statement that creates a check box named bold. The check box should be unselected, and it should have a “boldface type” label.

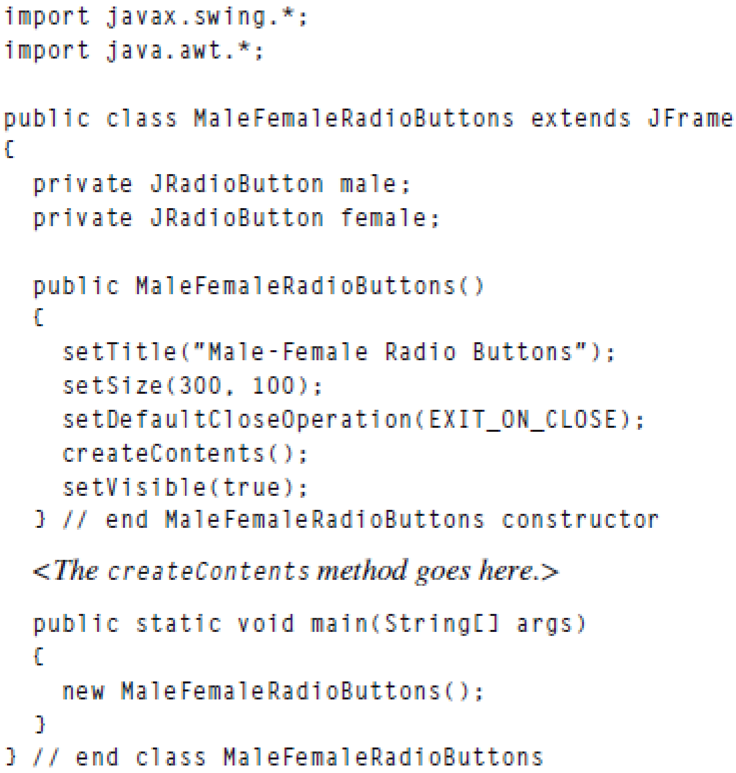
11. [after §18.13] How can your code determine whether a check box is selected or not?

12. [after §18.14] Provide a createContents method for a program that displays this window:



Gender:
● male
○ female

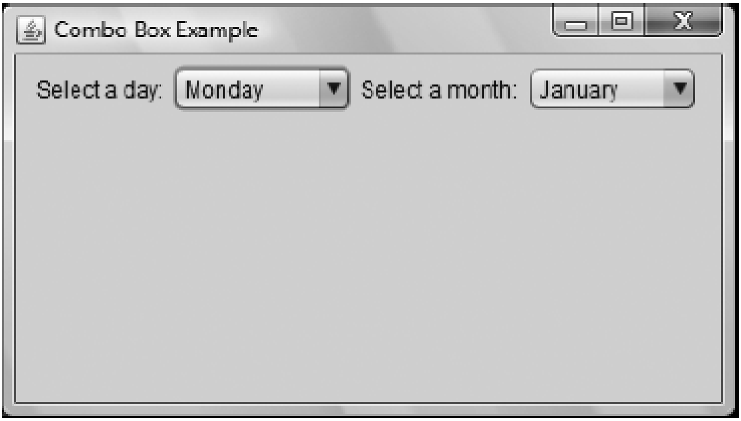
The male and female radio buttons should behave in the normal fashion—when one is selected, the other is unselected. Note that the male button is selected when the window initially displays. Your createContents method must work in conjunction with this program skeleton:



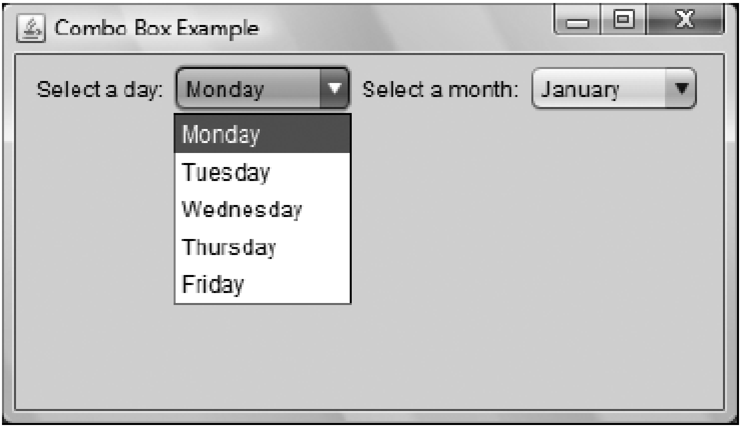
```
import javax.swing.*;
import java.awt.*;

public class MaleFemaleRadioButtons extends JFrame
{
  private JRadioButton male;
  private JRadioButton female;

  public MaleFemaleRadioButtons()
  {
    setTitle("Male-Female Radio Buttons");
    setSize(300, 100);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end MaleFemaleRadioButtons constructor

  <The createContents method goes here.>

  public static void main(String[] args)
  {
    new MaleFemaleRadioButtons();
  }
} // end class MaleFemaleRadioButtons
```

13. [after §18.15] The JCheckBox, JRadioButton, and JComboBox components are defined in what package?

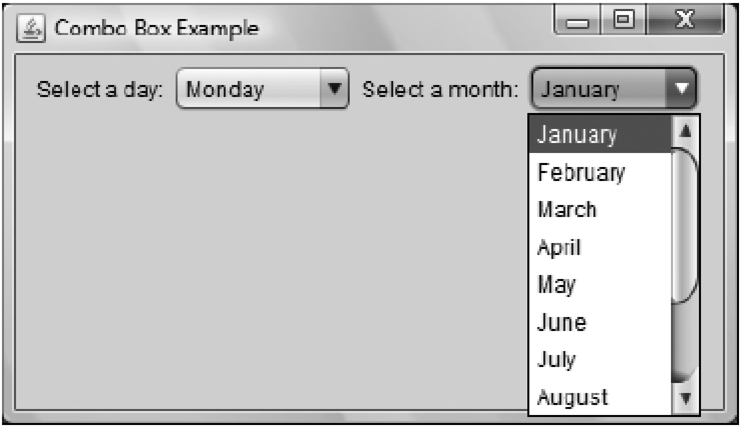
14. [after §18.15] Provide a createContents method for a program that initially displays this window:


Combo Box Example
Select a day:
Monday
Select a month:
January

When the user clicks the left combo box, this displays:


Combo Box Example
Select a day:
Monday
Select a month:
January
Monday
Tuesday
Wednesday
Thursday
Friday

When the user clicks the right combo box, this displays:


Combo Box Example
Select a day:
Monday
Select a month:
January
January
February
March
April
May
June
July
August

Your createContents method must work in conjunction with this program skeleton:

```
import javax.swing.*;
import java.awt.*;
public class ComboBoxExample extends JFrame
{
  private JComboBox daysBox;
  private JComboBox monthsBox;
  private String[] days =
    {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday"};
  private String[] months =
    {"January", "February", "March", "April", "May", "June",
     "July", "August", "September", "October", "November",
     "December"};

  public ComboBoxExample()
  {
    setTitle(“Combo Box Example”);
    setSize(400, 225);
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    createContents();
    setVisible(true);
  } // end ComboBoxExample constructor

  <The createContents method goes here.>

  public static void main(String[] args)
  {
    new ComboBoxExample();
  }
} // end class ComboBoxExample
```

Review Question Solutions

---

1. True.

2. Layout managers are in the java.awt package.

3. The FlowLayout manager places components left to right in a row until it runs out of space, and then it goes to the next row and does the same thing, and so on.

4. setLayout(new FlowLayout(FlowLayout.RIGHT));

5. frame.setLayout

(new BoxLayout(frame.getContentPane(), BoxLayout.Y_AXIS));

6. The five regions of a border layout are North at the top, South at the bottom, and West, Center, and East in a row between them.

7. True.

8. 0 or 1.

9. add(new JLabel("Stop", SwingConstants.CENTER), BorderLayout.CENTER);

or

add(new JLabel("Stop", SwingConstants.CENTER));

10. False. Specify both values, for rows and columns, only if you know the number of rows and columns in your table and the table is completely filled in (i.e., there are no empty cells). Otherwise, specify just one dimension that you're sure of and specify zero for the other dimension.

11. True.

12. Nothing. It does not change value.

13. JPanel containers are particularly useful with GridLayout and BorderLayout windows because each compartment in those layouts can store only one component. If you need a compartment to store more than one component, let that one component be a JPanel container and put multiple components into the JPanel container.

14. The add(new JLabel()); statement adds a dummy component (a blank label) to the bottom-left cell. The dummy component is necessary because, without it, the xLogPanel would go into the bottom-left cell, and that's inappropriate.

15. True. JTextArea components are editable by default.

16. False. JTextArea components do not employ line wrap by default.

17. If you click a check box that's already selected, the check box becomes unselected.

18. The following code creates a check box named attendance. The check box is preselected, and it has an "I will attend" label.

```
JCheckBox attendance = new
JCheckBox("I will attend",
true);
```

19. Nothing. It stays selected.

20. The clicked button becomes selected, and all other buttons in the group become unselected.

21. Combo boxes and radio button groups are similar in that they both allow the user to select one item from a list of items.

22. To determine the current selection for a combo box, call either getSelectedItem or getSelectedIndex.

23. If the radioGroup code is omitted from the JobApplication program, the program still compiles and runs, but the radio buttons operate independently. In other words, clicking one radio button will not cause the other one to be unselected.

24. This statement adds a 20-pixel blank margin to a JPanel container named panel:

```
panel.setBorder(new
EmptyBorder(20, 20, 20, 20));
```

25. JSlider constructor call:

```
new JSlider(0, 50, 10);
```

---

[1] Generating emails from a Java program is beyond the scope of this book, but the starting point is

installing Oracle's JavaMail package. To learn how to use the JavaMail package, go to http://www.oracle.com/technetwork/java/javamail.

[2] It may have occurred to you to insert the newline character, \n, into the component's text. Unfortunately, that doesn't work for JButton and JLabel components. However, it does work for the JTextArea component. We'll describe the JTextArea component later in the chapter.

[3] Here's the inside skinny. If you call the GridLayout constructor with two non-0 values for the rows and columns arguments, the columns argument is ignored and the GridLayout manager determines the number of columns on its own. For the case where you have two non-0 values and the table is completely filled, the GridLayout manager still determines the number of

columns on its own. But the determined number of columns works out perfectly (that is, the determined number of columns matches the specified number of columns).

[4] A draconian policy is a policy that is harsh or severe. The word *draconian* comes from Draco, a 7th-century B.C. government official from Athens who was in charge of codifying local law. Draco's laws were exceedingly severe. For example, even minor offenses were punishable by the death penalty.

[5] The Standard and Custom circles at the top of Figure 18.16 are called radio buttons. We'll describe JRadioButton components in the next section.

SYMBOLS

@Override annotation, 792

@ with command option, 954, 956–961

← left-pointing or backward arrow, 24–36, 41–44, 55–56

-> operator, 132–134, 330, 341, 809

:: operator, 566–568, 570–573

< > operator, 437, 910

! (factorial), 492

! (not) operator, 127, 131, 554
!= operator, 121, 127, 131
“” (double quotes), 99, 108
% (modulus) operator, 84
% symbol, with format specifier, 195
% with command option, 954
%% conversion specifier, 551
%= operator, 89–90
&& (and) operator, 124–128, 550–552
() (parentheses). *See* parentheses
* (asterisk)
  in directory tree, 771
  lines of, 68, 326–327, 333–334
  as multiplication operator, 35
  as multiplicity value, 594
  to set off block comments, 67–68
  as wildcard character, 176, 804

*= operator, 89–90
+ (plus sign)
  as addition operator, 35
  for concatenation, 72, 75, 93, 538
  in UML diagrams, 243
++ (increment) operator, 88–89, 537, 541–542, 544
+= operator, 89–90, 97
- (minus sign)
  as formatting flag, 197
  as subtraction operator, 13, 35
  in UML diagrams, 243
-- (decrement) operator, 88, 91, 543–544, 549, 556
-= operator, 89–90
/ (forward slash)
  in compound assignment operator, 86–87

as division operator, 13, 35
    in path directories, 768
    to set off comments, 67–68
/= operator, 89–90
; (semicolon)
    at end of do loop, 142–143
    for empty statements, 552–554
    required for Java statements, 13, 71, 74
< operator, 121
<= operator, 121
= (equals sign), 56, 75, 121
= pseudocode operator, 56
≠ pseudocode operator, 48
== operator
    avoiding for string comparisons, 129–130, 280–282
    enumerated type comparison, 558,

560
  - equals method and, 639–640, 642
  - in pseudocode, 56
  - purpose, 121, 279–280

> operator, 121

>= operator, 121

< > (angled brackets)
  - in ArrayList syntax, 437–440
  - in HTML tags, 737, 739–741, 825
  - for required descriptions, 128, 135, 149, 153

[] (square brackets)
  - ArrayList and, 443
  - in array declarations, 387
  - in arrays with two or more dimensions, 412–418
  - in for-each loop, 425–426
  - purpose, 70, 195

\ (backslash) character, 93–94

{} (braces)

  positioning, 70–71

  in switch statements, 133

|| (or) operator, 129, 551–552

0b prefix, 574

0x prefix, 574, 578

A

abbreviations, 73–74

abs method, 177

absolute get methods, 755

absolute put methods, 755

absolute paths, 767, 770, 772

abstract methods and classes, 565, 660–663, 754

Abstract Windowing Toolkit (AWT), 797

abstraction, 225

access modifiers, defined, 66. *See also specific modifiers*

access time, 6

accessing object data, 224, 252–253

accessor methods, 252–253, 333–334

acos method, 179, 181

ActionEvent class, 801–804, 807–809

ActionListener interface, 775, 778, 798

actionPerformed methods, 774, 783–784, 833

activity diagrams, 980–982

adapter classes, 799–803

add method, 460, 517, 519, 797, 805, 822–823, 826, 854, 856, 858

add-modules. *See* directives

addition

  its operator precedence, 86–87

  symbol for, 43–46, 86–89

address, 4

aggregation and composition

  compared with inheritance, 610

  defined, 592

  UML class-diagram representation, 611, 982–985

  using, 593–599

  using with inheritance, 610–613

Alert class, 107, 832–833, 851

AlertType class, 107

algorithms. *See also* pseudocode, programs, programs with GUI

  Asset Management, 55–59

  binary search, 406–407

  bowling score, 48

  CEO salary, 41–42

  defined, 11, 33

formats for, 33
fractal, drawing trees with, 517–523
growth modeling, 255–256, 260–261
if statements, 38–42
inputs, 36
Largest-prime-number, 49–50
Lightning, 39–40
looping structures in, 37, 43–50
operators and assignment statements, 35
output design, 33–34
performance evaluation of, 514–517
Prime Number, 49–50
Print Squares, 46–47
pseudocode varieties, 55–56
Rectangle, 34
recursive, expressed as functions, 500–502

- recursive binary search, 503–506
- sequential search, 403
- sorting, 409
- swapping values, 284–286
- Towers of Hanoi, 510–511
- tracing, 51–55
- variables in, 34–35

aliasing, 275, 284

alignment
- of BorderPane region contents, 871–872
- of FlowPane contents, 854

al-Khwarizmi, Muhammad ibn Musa, 30n

AmbientLight class, 682

American Standard Code for Information Interchange (ASCII). *See* ASCII values ancestor class, 602. *See also* inheritance

“and” operator, 124–128

angled brackets

  in ArrayList syntax, 437–438

  in HTML tags, 737, 739–741

  for required descriptions, 128, 135, 149, 153

animation, 933–936

anonymous functions, 808*n*

anonymous objects in an ArrayList, 446–450

API headings

  ArrayList methods, 439, 441

  basic features, 177, 178, 181, 186, 189, 201

API library

  AWT and Swing platforms, 789

  - ArrayList class, 436
  - Arrays.sort method, 410
  - Button class, 827
  - Calendar class, 368–370
  - character sets, 748
  - Collection interface, 465–466
  - equals methods in, 642–643
  - exception handling with, 708
  - LinkedList class, 451
  - overview, 171–176
  - recursion and, 518
  - shape-drawing classes, 677, 679, 682
  - use in bottom-up design, 359–361
- Apollo project, 363
- APPEND option, 749
- applets, 16
- application, 17, 107–108, 791–792

Application class, 25, 107, 790–792, 851

Application.launch method, 794

Application Programming Interface class library. *See* API library

arguments

- defined, 176, 230
- in main method, 70
- passing arrays as, 410
- passing in OOP, 250–251
- passing references as, 284–286

arithmetic operators. *See also* operators

- common types in algorithms, 35
- for numeric data, 83–85
- precedence of, 35–36, 86–87

ArithmeticException errors, 717

Arrays.copyOfRange method, 763–764

array methods, buffers, 756–760

arraycopy method, 394–396, 398–400, 506–507

ArrayDeque class, 454, 457–464. *See also* Collection interface

ArrayIndexOutOfBoundsException errors, 386, 717

ArrayList class, 437–442. *See also* Collection interface

  * multiplicity implementation, 596

  accessing elements, 439–440

  adding elements, 438–439

  API headings, 439, 441

  Big O performance, 510

  creating and using, 437–443

  versus LinkedList, 453–457

  methods in, 441

  overview, 436–437

  purpose, 176

- versus standard arrays, 450–451
- storing primitives in, 443–445
- updating elements, 440

arrays
- versus ArrayList, 450–451
- basic principles, 385–387
- copying, 393–396
- creating, 389
- declaring and creating, 387–390
- defined, 70, 157, 385
- element initialization, 389–390
- for histograms, 400–403
- length property, 391–393
- multidimensional, 418
- of objects, 418–424
- partially filled, 393
- polymorphism with, 654–660

  - searching, 403–408
  - shifting element values in, 397–398
  - sorting, 406, 408–412
  - two-dimensional, 412–418, 517

Arrays class, 176, 760

Arrays.sort method, 410

arrowheads, in UML diagrams, 984–985

arrow operator. *See* -> operator

arrow token. *See* -> operator

asCharBuffer method, 758

ASCII values
  - listed, 943–944
  - overview, 536–537
  - as Unicode starting point, 537, 574–576

asDoubleBuffer method, 752, 758–760

asFloatBuffer method, 758

asin method, 179, 181
asIntBuffer method, 752, 758–760
asLongBuffer method, 758
assert statement, 348–349
asShortBuffer method, 758
assignment
  compound, 88–90
  between data types, 79–80
  directionality of, 35–36
  embedded, 544–546
  versus equality, 121
  polymorphism with, 654–660
  in try blocks, 699–700
assignment statements
  for arrays, 389–390, 394, 413–414
  basic coding conventions, 75–76
  detailed analysis, 273–279

  embedded, 544–546
  promotion in, 539–540, 653
  for reference variables, 232, 273–279
  tracing, 53, 76–77
association, 158–159
association classes, 626–628. *See also* aggregation and composition
association lines, 614
asterisk
  in compound assignment operator, 89–90
  in directory tree, 771
  lines of, 68
  as multiplication operator, 35, 75
  as multiplicity value, 594
  to set off block comments, 67–68, 230

as wildcard character, 176, 770, 804

atan method, 179, 181

attributes, diagramming for UML classes, 227

autoboxing, 445

automatically imported classes, 176

auxiliary memory, 4, 6

averages, calculating, 47–48

AWT (Abstract Windowing Toolkit), 789–790, 952

azimuth, 649

B

back of a list, 452

backdoor, 185

background colors, 796, 816, 818, 820–822, 834–838, 840–841

backslash character, 93–95

backwards arrow, 55

base case, 475, 592, 602. *See also* stopping condition

base-16 number systems, 573, 643–644, 986–989

basic identities in Boolean algebra, 158–159

basic Latin subtable, 578

Beck, Kent, 370

behavior of objects, 223–224

Big O notation, 516–517

BigDecimal class, 535–536

BigInteger class, 535–536

binary number format, 5–6

binary numbers, 573–574

binary recursion, 497, 518

binary search, 406–408, 503–506, 517

binding. *See* dynamic binding

bins, 402

biological inheritance hierarchy, 599–600

bits, 5–6

blank lines

- after declaration statements, 327
- between code chunks, 75, 230, 329
- escape sequence for, 93–94
- excessive, 332
- between loops, 143–144
- for readability, 68

blank spaces, 332–333

block comments

- javadoc, 336–337
- syntax, 67–68

block, 122, 244, 329–331

Blu-ray Discs, 8

body of loop, 46, 48

Boolean logic, 152–160

Boolean methods, 253–255

Boolean values, 120–121, 152, 156

Boolean variables

- default value, 237
- validate method and, 289–290
- for input validation, 156–157
- when to use, 153

Border class, 216, 218, 221

BorderPane container, 851–852, 867–871

- areas, 867
- border properties, 869–872
- components, 868

BorderStroke class, 216, 218, 221

border properties, 815–816, 818, 820–822, 834, 836

borders of containers, 851–852, 867–

871

borders of nodes. *See* border properties

bottom-up design, 359–361

Box class, 677, 681–683

BoxBlur class, 931–935

braces

  positioning, 70–71, 329

  for subordinate statements, 122–123, 330

  in switch statements, 133

breadth-first traverse, 475

break statements

  in looping structures, 554–556

  in switch statements, 138–139

breaking points for long statements, 330

Buffer class, 752

Buffer methods, 754–755

- absolute and relative put methods, 729
- array methods, 756–760
- ByteBuffer methods, 728–729
- character sets, 760

buffered text file I/O, 749–752. *See also* Files class

BufferedReader class, 752, 753

BufferedWriter class, 749–751

BufferOverflowException errors, 755

buffers

- defined, 749
- primitive, 752–760
- StandardOpenOption, 749–751, 761, 764–765

Bug class in GridWorld software, 619–621

BugRunner driver for GridWorld software, 579–580, 619n, 620–622 bugs, 51. *See also* tracing

Button control, 796, 826–827, 851

buttons

- overview, 788–789, 826–827
- methods, 827
- styling, 829
- event handling, 831–832

byte, 5

byte array methods, 756–760

ByteBuffer class, 752

ByteBuffer methods, 754–756

bytecode, 15, 73

C

C++, 12, 16

calculator (Windows), 644

calculator division, 83–84

Calendar class, 175, 368–370

call stack, 495

call stack trace, 715, 717

calling objects

  defined, 171

  identifying, 235–237, 292

camelCase, 35

capacity, 5

capital letters

  in class names, 69, 73

  converting to, 186, 188–189, 192–193

  identifier coding conventions, 73

  ignoring in string comparisons, 100

  for named constants, 297

  in variable names, 35

Cascading Style Sheets (CSS), 789, 812, 823

case label arrows, 132–133

case label colons, 138

case labels and clauses, 132–139, 323

case conversion, 192–193

case sensitivity, 66

case-based design, 361

cast operators, 90–92, 539, 540–541

catch blocks

  generic, 708–709

  for input validation, 697–698

  multiple, ordering, 712–714

  overview, 692–693

  postponing, 718–720

CategoryAxis class, 725, 727

CD-ROMs, 8

central processing units, 2, 3–4

chained method calls, 142, 273, 286–289

ChameleonCritter and

ChameleonRunner classes in GridWorld software, 596

char data type
  ASCII values, 536–538
  cannot assign integers to, 541
  concatenating, 538
  overview, 93
  as primitive type, 95

Character class, 186

character sets, 748–749, 760

characters
  determining position in strings, 191
  permitted in identifiers, 73
  underlying numeric values, 536–537

charAt method, 98–90, 124–125, 135–137, 141–142, 149, 151

Charset class, 748–750

CheckBox control, 851, 900–901, 903–

904

check box, 897

checked exceptions, 700–702, 705–707

Page 994

child class, 602

chips, 4

ChoiceDialog class, 107, 476–481, 777–778, 851

Circle class, 851, 890, 928–935

circular array, 458

.class extension, 70

class diagrams, 982–985. *See also* UML class diagrams

class headings, 69

class reserved word, 69

classes. *See also h*ierarchies

- defined, 69, 225
- enumerated type as, 561–563

- naming rules, 73–74, 329
- organizing within programs, 592, 613–618
- relation to objects, 26, 171, 225–226
- selecting in top-down design, 350

cleanup code, 720, 722, 723

clients, 338

clock speed, 4

close-window buttons, 25, 795

closing braces, 71. *See also* braces

cloud storage, 7

code displayed in this book. *See* programs

code with GUI displayed in this book. *See* programs with GUI

code reusability, 602

coding-style conventions, 325–334

col variable, 75

Collection interface, 464–468

Collections class, 176, 465, 468

Color class, 204–205, 220, 623, 632, 851, 931–935

colors

- creating gradients of, 204–206
- GUI overview, 834–838
- color names, 835

columns

- in GridPane, 854–858
- in tables, 763

ComboBox control, 851, 909–913

combo boxes, 897

COMDEX, 10

command option, 954

command prompt windows, 22

commas

- as flag characters, 197–198

  - omitting from long numbers, 76
  - separating variables with, 74
- comments
  - aligning, 75
  - for blocks and obscure statements, 332
  - forms of, 67–68, 326
  - for methods in OOP code, 230, 221
  - with variable declarations, 74, 330
- communication objects, 343
- commutation, 158–159
- compact disc, 8
- compareTo method, 188
- comparing enumerated type values, 558–561
- comparison operators, 121. *See also* operators
- compilation, 15, 23, 73

compilation error, 80
compile commands, 13
compilers, 14, 21–22, 67n
compile-time errors
  with abstract classes, 662
  defined, 80, 131, 699
  examples, 700
  expressions producing, 550
  lambda-expression local variables, 861–862
  with inheritance, 653, 657, 658
  with overriding methods, 647
  with toString method, 647, 651
components, 789
  adding to BorderPane containers, 868
  adding to GridPane containers, 854–858

  in composition, 592
  as GUI elements, 789–790, 796–797
Composite design pattern, 347
composites, 592
composition, 592–599. *See also* aggregation and composition
compound assignment operators, 88–90
compound condition, 124–125
compound statement, 122, 133
computer ethics, 18
Computer Ethics Institute, 18
computer hardware, 2–10
computer improvements, 10
Computer Professionals for Social Responsibility, 18
computer programs. *See also* program design

compiling source code into object code, 13–14

defined, 2, 10

portability, 14–15

source code, 12–13

steps in creating, 10–12

computer system, 2

concatenation

of char and String, 93, 538, 548

evaluating expressions, 548

of strings, 75, 97

concerns, separating, 346

condition component of for loop, 145

conditional operator expressions, 546–547

conditional structures, 37, 248. *See also* if statements

conditions

  defined, 37

  in if statements, 120–128

  in looping statements, 140, 142–143, 145–147

consistency in GUI design, 812, 821, 898, 915

constants

  basic types, 80–82

  in interfaces, 665–668, 673

  in Math class, 180–181

  using, 82, 83

  wrapper class, 182

constant-width characters, 576

constructors

  with arrays of objects, 419

  benefits of, 293–294

  coding conventions for, 889

- default, 295–296, 603–604, 606
- defined, 293–294
- elegance of, 299
- grouping in code, 333–334
- instance constants with, 296–297
- overloaded, 299–303, 606
- in super- and subclasses, 605–606

Consumer argument, 564, 566. *See also* Consumer interface

Consumer interface, 565–566, 570

containers. *See also* layout panes.

- defined, 797, 801, 818, 849–852
- inheritance relationships, 851
- management role, 797, 852

continuation lines, 330

continuous exponential distributions, 200, 201

continuous uniform distributions, 199–

200

control, 789, 851

controller, 521–522. *See also* Model-View-Controller (MVC)

control characters, 94, 537

Control class, 789, 796–797, 851

control statements

  Boolean logic, 157–160

  boolean variables in, 152–157

  conditions in, 120–121

  do loops in, 142–143

  for loops in, 144–146

  if statements in, 121–123

  input validation, 156–157

  logical operators in, 124–127, 129, 131

  loop structure selection, 149–150

  nested loops in, 150–152

  purpose of, 120
  switch, 132–138
  while loops in, 138–140
conversion
  evaluating, 548–550
  primitive types to objects, 182–186
  by promotion, 539–540
  of upper- and lowercase characters, 186
  using cast operator, 90–92, 540–542
conversion characters, 196
coordinate system, 678
copying arrays, 393–396
copying files, 768
CornerRadii class, 218, 221
cos method, 179, 181
count variables
  basic function, 43–44

  terminating loops with, 46

countdowns, 144, 146, 556–557

CPUs (central processing units), 3–4, 10

crashes, 99. *See also* error messages, exception handling

CRC cards, 370–374, 476–482, 775–780

createContents method, 766, 783

Critter and CritterRunner classes in GridWorld software, 619–622

cryptic code, 546

CSS properties, 816–818

CSS rule. *See* stylesheet

Cunningham, Ward, 370

curly braces. *See* braces

current directory, 23

customer requirements, 10, 11, 30

cylinder, 677–680

CycleMethod class, 204–206, 931–935

Cylinder class, 677–680

D

-d option with javadoc, 335

data files. *See* files

data in objects, 224–226, 228

data types. *See also* return types

  for constants, 80

  converting, 90–92, 182–186

  in declaration statements, 74

  ignoring in pseudocode, 35

  in initialization statements, 77–78

  numeric, 78–80

  in overloaded methods, 289

  as return values, 247–248

  specifying in class diagrams, 227–

228

database, 17

days of the week enumeration, 557

debuggers, 55, 242. *See also* tracing

debugging with temporary print statements, 252

decimal points, 78

declaration statements

- analysis, 274
- arrays, 387–390
- basic syntax, 74
- coding conventions for, 327
- field, 314, 327
- for serialization, 743–744
- static constants, 313, 314
- static variables, 304, 314
- variables as interface types, 452–453

decrement operator, 88–90, 543–544

default character sets, 748, 750–753

default constructor and call to, 295–296, 603, 606

default methods, 665, 672, 674

default values

- array elements, 390
- for GUI nodes, 794, 796, 799, 801, 819, 821, 838
- instance variables, 236–237
- reference variables, 236–237, 274
- static variables, 305

delays, creating with empty statements, 552

deleting files, 768

DeMorgan's theorem, 158–159

depth-first traverse, 771

derived class, 602

descendant class, 602

design. *See* program design

design patterns, 346–347

design philosophy, 345–349

desktop computers, 7, 17

development environments, 19

dialog boxes. *See also* Graphical User Interfaces (GUIs)

  defined, 20, 107, 851

  displaying, 108–111

diamond operator, 437–438

diamonds in flowcharts, 37, 120

digits, 572–573

directives (for compiling and execution)

  add-modules, 954

  exports, 948, 952, 960

  path, 954

- requires, 952–953

directories, 20, 768–769

directory tree search, 769–774

discrete triangular distributions, 200, 201

discrete uniform distributions, 199–200

disk drives, 7–9

disk space, 9

diskettes, 9

display, 3

distribution, 158–159

division

- floating-point, 83–84
- integer, 84–86
- its operator precedence, 86–87
- by zero, 48, 301, 687

do loops

  - overview, 142–143
  - when to use, 149–150
- documentation
  - defined, 10, 363
  - for outsiders, 334–338, 344–345
  - self-documenting code, 74
- DOS operating system, 360–361
- double constants, 80
- double data type
  - converting to strings, 182
  - default type, 536
  - default value, 236–237
  - as primitive type, 95
  - when to use, 78–79, 533–534, 536
- double quotes, 71, 93–95, 189–190
- Double wrapper class, 183
- Double wrapper objects, 443–445

DoubleBuffer class, 752, 758, 760

doubleValue method, 445

doubly linked elements, 451

Dow Jones Industrial Average, 398, 443

drawOccupants GridWorld method, 619

drawing lines, 216–221, 724–728

drawing rectangles, 216–221

driver classes, 231–234

drives, 7

dropdown lists, 909

dummy constructors, 296

dummy methods, 658–661

dump, 420

duplicate code, avoiding, 301

duplicate names. *See also* naming rules

  in overloaded methods, 289–293

  - using inheritance to eliminate, 611
  - for variables in separate blocks, 251

- DVDs, 8
- dynamic allocation, 304
- dynamic binding, 649–651, 653–654

E

- E (Element) return type, 439–440
- echo printing, 135
- editing text, 19. *See also* setEditable method
- elegant code
  - basic features, 131
  - with constructors, 299
  - with loops, 146, 154
- elements. *See also* arrays
  - accessing in arrays, 385–386
  - in array declarations, 385–386

  - ArrayList handling, 437–443
  - defined, 436
  - doubly linked, 451
  - generic class, 440
  - initializing, 389–390
  - shifting, 397–398
- ellipse, 206
- Ellison, Larry, 17
- else body, in recursion, 492
- else clause, in recursion, 506, 511
- else if, 38, 40–41, 123
- embedded assignments, 544–546
- embedded driver, 363–365, 448–450
- embedded panes, 882–887, 889, 913
- empty statements, 248–249, 552–554
- empty strings, 190, 192
- encapsulation

  - basic principles, 224–225
  - of helper methods, 338–342
  - implementation techniques, 342–343
  - variable names and, 251
- end comments, 332
- end tags, 741
- end-user needs, 33, 344–345
- enum reserved word, 25, 557
- enum types, 133n, 557–564, 850*n*
- enumerated types
  - as a class, 561–563
  - comparing values, 558–560
  - days of the week, 557
  - letter grades, 558, 559–561
  - retrieving all objects in, 563–564
  - values method, 563
- EOFException class, 746–747

EOFException errors, 746–747, 777

equality

  versus assignment, 121

  testing objects for, 279–284, 639–640

equals method

  implementing, 280–284

  programmer-defined, 640–642

  purpose, 99, 130

  syntax and semantics, 639–640

equals sign, 56, 75, 121

equalsIgnoreCase method, 100, 130, 283–284

error messages. *See also* exception handling

  Alert.AlertType icon for error, 107

  analyzing, 714–718

  compilation, 22
  information in, 99
  from static method, 310–311
escape sequences
  basic features, 93–95
  Unicode, 575–576
ethics, 18
Euler's number, 180
evaluating expressions, 86–88
evaluation, recursive methods, 500–503
event handlers, 789, 801, 804–809
event-delegation model, 806
event-driven programming, 805–807
events, 805, 807, 832–834
example programs. *See* programs
example GUI programs. *See* programs with GUI

Exception class getMessage method, 709

exception handling

- approaches for checked exceptions, 705–707
- approaches for unchecked exceptions, 702–705, 714
- documentation, 708
- exception categories, 700–702
- with generic catch block, 709
- with multiple catch blocks, 712–714
- overview, 692–693
- postponing catch, 718–720
- try and catch block advantages, 695–698
- try and catch block overview, 693–695
- try block details, 699–700
- understanding error messages, 714–

    718
Exception objects, 693
exceptions
    categories of, 700–702
    defined, 386, 693
exclamation point
    as logical operator, 131
    as factorial symbol, 492
executable program, 67
execution time comparison, methods, 453–457
exponential growth, 255
exponents, 534–535
exports. *See* directives
expressions
    casting, 92
    defined, 83
    evaluating, 86–88, 547–550

extending an interface, 465
extends clause, 604
external drives, 7–8

F

F suffix, 81
factorial method
    Big O performance, 517
    iterative, 496
    recursive, 493–495
factorials
    finding with recursion, 492–495
    with for loop, 147
fall-through mechanism, 138
false reserved word, 120–121
Fibonacci sequence, 502–503
FICA tax calculation, 673–676
field, 180, 314, 327. *See also* constants,

static variables, instance variables

file

  accessing, 9, 748–749

  accessing with GUI dialogs, 775–780

  creating, moving, copying, and deleting, 768

  defined, 9, 736

  HTMLGenerator example, 737–741

  input/output approaches for, 736, 743–747, 760–767

  memory-mapped, I/O and, 760–767

  paths, 767–768

  reading a serializable object from, 745–747

  storing Java objects in, 743–744

  writing a serializable object to, 744–745

FileAlreadyExistsException errors, 708, 768

FileChannel methods, 760–767

FileChooser class (dialog box), 777–779

Files.Exists method, 706, 708, 768

FileIn.hasNext method, 681

FileInputStream class, 746, 779–780

FileNotFoundException errors, 712–714

FileOutputStream class, 744–745, 777–778

FileSystem and FileSystems classes, 773

Files class. *See also* buffered text file I/O

- Files.createFile method, 706–707, 768
- Files.get method, 767
- Files.isDirectory method, 768
- Files.isRegularFile method, 768

Files.newBufferedWriter method, 750–751

FileVisitResult class, 773–774

filters, 253

final modifier

with methods and classes, 610

with named constants, 81, 297, 312

prohibited with abstract methods, 663

finally blocks, 720, 722

firing an event, 763

first-in first-out (FIFO), 457–460, 468, 749

fixed seed, 203

flag signals, 491

flags, 195–197

flash drives, 8

flash memory, 8

flip method in Buffer classes, 754–757

float data type

  default value, 237

  as primitive type, 95

  when to use, 78–79, 534

FloatBuffer class, 752, 758

Float wrapper class, 182

floating-point casts, 92

floating-point constants, 80

floating-point numbers, 78

  division of, 83–84, 92

  initializing, 236–237

  when to use, 78–79, 534, 535–536

floppy disks, 9

flow of control, 37–38. *See also* control statements

flowcharts

  - CEO Salary, 42
  - defined, 33
  - Happy Birthday, 44
  - to illustrate flow of control, 37–38
  - with if statements, 42

- FlowPane container, 372–373, 797, 801–805, 851–852, 853–854
- folders, 20–21
- font, 26–27, 576–577, 791–794
- for loops
  - creating delay with, 552
  - versus for-each loops, 450
  - headers, 555–557
  - index variables, 146–148, 151, 244
  - nested with two-dimensional arrays, 413
  - overview, 144–146
  - when to use, 146

for-each loops, 425–426, 446, 449–450

forEach method, 564–567

formal pseudocode, 284–285

format specifier, 189, 195–197

format string, 189, 195, 197, 417–418

formatting characters, 19

formatting output, 194–199

forward branching statements, 138

fractal algorithm, drawing trees with, 517–523

fractal, 518

fractional digits, 196

free software, 361

Free Software Foundation, 361n

free space in memory, 279

freezing random number sequences, 203

frequency distribution, 400

functional interface, 565, 570, 573

functional method, 565

functions

  mathematical, 69–70, 177, 467

  recursive algorithms expressed as, 500–502

fx.txt file, 954

G

garbage collection, 276–279

garbage values, 90, 245, 274

Gates, Bill, 10

Gemini project, 363

general-purpose swapping algorithm, 284–285

generic catch blocks, 709, 714

generic classes, 438, 440

get methods

  absolute, 755

  - ArrayList, 439–441, 444–445, 450, 452–456, 517
  - Buffer, 752, 755
  - CalendarDemo class, 369
  - getStylesheet, 822–825, 830, 840
  - defined, 252–253
  - LinkedList, 452–456, 517
  - relative, 755–756
- getBytes method, 759–760, 764
- getChar method, 755
- getClass method, 709–711, 713, 723, 726
- getEventType, 924–926
- getInputStream method, 741–742, 752
- getInstance method, 368
- getMessage method, 709
- getSelectedItem method, 916
- getSize method, 256

getSizeIncrement method, 256

getText methods, 798, 800, 804, 808, 827, 831–833

.gif files, 619, 923–924

gigahertz (GHz), 4

Gosling, James, 279

graceful failures, 148

grade school division, 84–85

Graphical User Interfaces (GUIs). *See also* programs with GUI

  Alert dialog box, 107–108, 111, 833

  ChoiceDialog box, 107–110

  TextInputDialog box, 107–109

  basic input/output implementations, 107–111

  Unicode characters, 575–579

grayed out, 800, 900

Green Project, 16

GridPane container, 851–857, 854–858

GridWorld case study, 579–580, 619–622. *See also* programs with GUI

  Actor images, 619–620

  Actor behavior, 620–621

  Critter class, 621

  CrabCritter class, 596

  CrabRunner class, 622

  CritterRunner class, 622

  extensions of Critter class, 622

grouping constructors, mutators, and accessors, 333–334

Group class, 204–205, 520. *See also* layout panes

growth methods, 256, 260–261

growth modeling, 255, 259

GUI programs, 24–27.

See also Graphical User Interfaces (GUIs), programs with GUI

GUI windows, 107

H

Happy Birthday algorithm, 43–46, 52

hard disks, 3, 7–10

hardcoded constants, 81

hardware, 2–10

Harvard Mark II, 51

“has-a” relationship, 593, 610, 616

HashMap, 468, 471, 517. *See also* Collection interface

hash table, 644

hashcode values, 643

hasNextLine method, 739, 752

HBox container, 851, 852, 882–887. *See also* layout panes

headings

- for API methods, 176–179, 181, 186, 189
- class, 69
- for loops, 555–557
- HTML, 737, 739–740
- main method, 69–70
- return value types in, 247–248

heads, 7

heavyweight components, 790

helper methods
- implemented in top-down design, 354–356
- of main method, 312
- overview, 338–342

Hertz, 4

hexadecimal numbers
- for ASCII characters, 943–944
- conversions, 986–989

  - for hashcode values, 643
  - overview, 573–574
- hexits, 573
- hidden characters, 19
- hierarchies
  - assignment between classes within, 653
  - combining approaches, 611–613
  - composition and aggregation, 592–599
  - Exception class, 701–702
  - inheritance examples, 603–610
  - inheritance overview, 599–603
  - packages, 951–952, 955–960
  - Java Collections Framework, 464–468
  - polymorphism with explicit inheritance, 654

high-level pseudocode, 56
histograms, 400–401
home-appliance software, 15–17
Horton’s Law, 177
hot swapping, 8
HTML tags, 737, 739–741
human body, 593
hyphens
  as flag characters, 197
  as subtraction operator, 13
  in UML diagrams, 243

I

i icon, 107
I/O devices, 2
icons for information dialogs, 107
identifier naming rules, 73–74
“if, else” form of the if statement, 40,

122, 131

"if, else if" form of the if statement, 40–41, 123, 132, 138

if statements

  basic forms, 38–41, 122–123

  braces with, 331

  versus conditional operator code, 547

  equals methods in, 281–284

  overview, 121–122

  recursive stopping condition, 491–494

  return statements and, 249

  shortened, 254

  versus switch statements, 132, 138

IllegalStateException errors, 710–711

Image class, 204–205, 851, 923–924

ImageView class, 204–205, 851, 923–

924

immutability of string objects, 192

implementation
- of computer programs, 57–59
- defined, 11
- of interfaces, 452, 465–467, 663–668, 670–672
- of a recurrence relation with a loop, 502–503
- implements clause, 664

implicit method calls, 644–647

import statements
- ArrayList class, 437
- for Scanner class, 101
- wildcards in, 176, 368

inaccessible objects, 279

increment operator, 88–90, 541–544

indentation

  with braces, 122, 329–330
  with line wrap, 330
  in pseudocode, 39
index parameter, 755
index position, 98, 189, 191
index variables
  defined, 146
  multiple, 556–557
  in nested for loops, 147
  scope of, 148, 244
indices, for arrays and lists
  basic principles, 385–387
Page 1000
  for combo boxes, 913
  defined, 385
  invalid, 704–705, 718–720
  with two-dimensional arrays, 412–413

indexOf methods, 189, 191

IndexOutOfBounds Exception errors, 704–705, 718–720

infinite loops, 45, 55, 142, 495

INFORMATION dialog-box icon, 111

inheritance

- with aggregation and composition, 610–613
- assignment between classes with, 653
- versus association, 628
- Buffer methods, 754–756
- versus composition, 616–619
- defined, 592
- equals method, 639–643
- overview, 599–603
- polymorphism within arrays, 654–660

  - sample implementations, 603–610
  - scene graph, 819
  - toString method, 643–644
- IntBuffer class, 727
- initialization
  - array elements, 381–382, 412–413
  - ArrayList, 437
  - assigning values during, 48, 50, 77, 78, 81
  - combining with instantiation, 293
  - field, 327
  - of named constants, 297
  - in try blocks, 699–700
- initialization component of for loop, 145
- initialization statements
  - basic syntax, 77
  - blank line after, 329

  - garbage values in, 90
  - for instance variables, 229–237
  - in try blocks, 699–700
- inline style, 812
- input devices, 2
- input dialogs, 107–111
- input statements, 53
- input validation, 156–157, 697. *See also* exception handling
- InputMismatchException errors, 694–695
- InputStream class, 741–742
- InputStreamReader class, 752
- input/output, types of, 736
- input/output operations
  - buffered text files, 749–752
  - channel I/O and memory-mapped files, 760–767

  - creating, moving, copying, and deleting files, 768
  - directory content description, 768–769
  - directory trees, 769–774
  - file access, 748–749
  - HTMLGenerator example, 737–741
  - major approaches, 736–737
  - object-based, 743–747
  - path definition and manipulation, 767–768
- inputs
  - into algorithms, 36
  - converting to Strings, 647–648
  - Scanner class features, 100–105
  - testing, 347
- InputStream class, 741–742, 752
- insertions into strings, 193

insertion sort, 515–517

Insets class, 218–219, 869–871

instance constants, 297–299, 312–313, 314

instance fields. *See* instance constants, instance variables

instance methods

  calling, 235–236, 310–311

  with static methods, 310–311

  defined, 171, 226

instance named constants, 312–313, 314

instance variables (fields)

  accessing without a this reference, 365

  in containing classes, 596

  copying, 275–276

  declaring, 228–230

  - default values and persistence, 236–237, 274
  - defined, 226
  - encapsulation with, 342–343
  - for defining class relationships, 614–616
  - versus local variables, 244
  - public modifier and, 228, 743
  - text box components as, 780
- instanceOf operator, 652–653
- instances, objects as, 226
- instantiation
  - arrays, 389, 419
  - combining with initialization, 293
  - defined, 233, 274
  - LinkedList class, 451–452
  - objects with same instance variables, 275–276

- temporary objects, 276–279

instruction sets, 14n

int cast operator, 92, 179, 185

int constants, 82

int data type
- converting strings to, 182–185, 697–698
- default value, 236–237
- dialog input, 105–106
- as primitive type, 95
- when to use, 78

intBuffer class, 752, 758, 760

Integer data types, 532–533

Integer class, 182, 532–534, 696–701

integers
- assigning to characters, 541
- dialog input, 105–106
- division of, 84–85

  - initializing, 236
  - when to use, 75, 532–533
- integrated development environments, 19, 242
- intellectual property protection, 18
- intelligent appliances, 15

Page 1001

- interfaces
  - Collection interface, 465–466
  - extending, 465
  - implementing polymorphisms with. *See* multiple inheritance.
  - inheritance, 664
  - List, 452–453, 465–466
  - Map, 465–467
  - Queue, 459, 460, 469
  - uses, 663–673
  - with outside world, 339, 788, 797,

  799, 826

interface reserved word, 665, 949

invalid input, 103

InvalidPathException errors, 708, 709, 711, 713, 714, 725

I/O devices, 2

IOException class, 606–708, 702, 706–708, 710–712, 714, 741, 749, 763–765, 770, 773–774

“is-a” relationship, 610, 616, 626

isDigit method, 186

isEmpty method, 189–190, 459, 465, 472

Iterable interface, 566–567

italics for descriptions, 38

iterating, 361–363

iterations

  confirming number of, 46

  defined, 45
  with for-each loops, 426
  with for loops, 144–148
iterative enhancement, 325, 361–363
iterative solutions versus recursive, 496–499
iterators, 457
intractable, 517

J

Java
  emergence of, 15–18
  character sets in, 748
  portability problem solution, 14–15
  statements, 13
Java API library. *See* API library
Java API website, 172–175, 359–360
JavaFX

  definition, 790
  libraries, 24
  path to, 22
  classes, 851
Java collections framework
  collections example, 468–475
  overview, 464–468
Java Development Kit (JDK)
  installing, 22
  javadoc tool, 335–338, 363
Java Enterprise Edition (Java EE), 16
.java extension, 73
Java Micro Edition (Java ME), 17
Java programming language, 13, 17, 20
Java Server Pages (JSP), 16
Java 7, 24
Java Standard Edition (Java SE), 17

Java Virtual Machine (JVM), 15, 83–84, 235, 445

javac command, 23

javadoc tool, 335–338, 363

java.io package, 749, 751, 752

java.lang package, 176

java.nio package, 752, 758

java.nio.file package, 749, 767, 769, 771

java.util package, 175–176, 437, 453, 464

java.util.LinkedList, 456

javax.swing package, 749

.jpg files, 205, 219

K

K&R formatting, 822–823

Kennedy, J. F., 362

key, map, 467

KeyFrame class, 934–936

KeyValue class, 934–936

keySet method, 467

keywords, 69, 947–950

kludges, 660

L

Label component, 26–27, 791–792, 851

lambda expression, 227, 565, 807–808, 856, 862, 881

laptop computers, 7

last-in first-out (LIFO), 460

late binding, 650. *See also* dynamic binding

layout panes, 851, 852–853. *See also* containers

  BorderPane. *See* BorderPane container

  embedded, 882–889, 890

  - FlowPane. *See* FlowPane container
  - GridPane. *See* GridPane container
  - Group. *See* Group class
  - HBox. *See* HBox container
  - Pane. *See* Pane container
  - SplitPane. *See* SplitPane container
  - StackPane. *See* StackPane container
  - TextFlow. *See* TextFlow container
  - TilePane. *See* TilePane container
  - VBox. *See* VBox container
- leading whitespace, 102, 105
- legacy code, 137
- length method, 99, 171, 391–393
- length property, 391–393, 411, 412, 413–414
- letter grades, 558–561
- lexicographical ordering of strings, 188–189

lightweight components, 790

limit method, 754–755, 761

limited-resource devices, 17

LineChart class, 724–727

line breaks, 330

Line class, 518–520

line drawing, 216–220, 722–728

line numbers, 52–54

line plotting, 695–698

linear recursion, 497

LineChart class, 724–728

LinkedHashMap class, 476–482. *See also* Collection interface

LinkedHashSet class, 470–473. *See also* Collection interface

linked classes, 601

linked lists, 436, 451–452

LinkedHashSet class, 436, 465–473

LinkedList class, 176, 436, 451–452

  versus ArrayList, 453–457

Linux, 19, 20*n*, 22

List interface, 452–453, 465–467

list parameter, 453

listener, 806

List. *See also* ArrayList, arrays

  linked, 436, 451–452

Literals, 81, 93

local main methods, 365

local named constants, 297–298, 312, 314

local variables

  basic features, 244, 274

  defined, 240, 244

  encapsulation with, 342–343

  versus parameters, 250

    persistence, 247
    temporary, 276, 286
    using, 244–247
    when to declare, 804–805, 861, 879
local variable type inference, 148, 227
logic errors, 130–131
logical chunks of code, separating, 75
logical operators. *See also* operators
    “and” operator, 124, 127–128
    “not” operator, 131
    “or” operator, 124, 129
    in Boolean algebra, 157–159
logistic equation, 256, 518
long data type
    default value, 236, 237
    as primitive type, 95
    when to use, 78, 532–535

long-form tracing, 52–54

LongBuffer class, 752, 758

Long wrapper class, 182

lookup method in TextArea class, 902. *See also* TextArea control

look and feel, 789

looping structures

- algorithm forms, 37, 43–45
- assignment statements in, 545–546
- braces with, 331
- break statement within, 554–555, 556
- choosing, 149–150
- do loops, 142–144
- flow of control in, 48–50, 142–143
- for loops, 144–148, 556–557
- infinite recursive, 495
- insertion sort, 515, 517

  nested, 45–48, 150–152
  recurrence relation implementation, 502–503
  return statements in, 249
  in sorting algorithms, 409–410
  termination techniques, 45–47
  while loops, 138–142
Lovelace, Ada, 104
lowercase letters
  converting to, 186, 189, 192
  in variable names, 35
Lucas, Édouard, 510

M

Mac OS X, 19
machine code, 14
main memory, 4–6
main method

  - in driven classes, 363–385
  - headings, 69–71
  - placing variable declarations in, 74
  - throws Exception, 105–106
  - with JavaFX, 791–792, 794
- maintenance, 10, 249, 363
- makeCopy method, 276–277
- mammals, 599–600
- Map interface, 465. *See also* Collection interface
- map method, 762–763, 765–766
- maps, 464
- mark method, 754–755
- matching catch blocks, 695
- Math class, 171, 176, 177–181
- MathContext class, 535–536
- mathematical functions, 69–70, 167, 467

Math.random method, 178–179, 184, 199–201, 214–215

Math.round method, 171, 178–179, 181, 226

maturation, modeling, 255–256

maximum value

  between two numbers, 546

  for integers, 532

  named constants for, 532, 533

  for random number generation, 199, 202

meaningful names, 329

megabytes, 9

members, 227

memory (computer), 4–8, 279, 304

memory-mapped files, I/O and, 760–767

memory leaks, 279

MenuBar class, 851, 919–921
MenuItem class, 851, 919–921
menu bars, 918
Menu class, 796, 851, 919–921
Mercury project, 362
message dialogs, 107–111
message propagation, 470, 473
method body, 229–230
method overloading, 289–293

method overriding
    defined, 592, 606
    implementing, 606–608, 771, 774
    toString methods, 644
method reference, 565–567, 572, 807–808
method signatures, 289
method-call chaining, 273, 287–289

method-call prefixes, 310–311, 342

methods

  basic math, 177–179

  Character class, 186–188

  defined, 37, 69–70

  description format, 327

  execution time comparison, 453–457

  multiple-method programs, non-OOP and, 213–216

  naming, 73, 329

  overloaded, 289–293

  promotion in calls, 540

  recursive, guidelines, 491–492

  relation to objects, 171, 224

  static versus instance types, 171, 226

  String, 97–100, 188–193

  trigonometric, 179, 181

  wrapper class, 182–183

Micro Edition applications, 16
microprocessor, 3
Microsoft, 9–10
Microsoft Windows operating system, 360–361
middleware, 17
MIN_NORMAL constant, 533
MIN_VALUE constant, 532
minimum values
  for integers, 532
  named constants for, 532, 534
  for random number generation, 199
minus sign
  as subtraction operator, 13, 35
  as unary negation operator, 86–87
misuse of empty statement, 553–554
mixed expressions
  defined, 84, 539

  - evaluating, 548–550
  - promotion in, 84, 539–540
- Modality class, 477–479
- modal component, 833
- Model-View-Controller (MVC) design pattern, 346, 517
- modifiers, 78
- modular organization, 338–342
- modules
  - defined, 174, 952
  - module-info program, 952
  - module descriptor, 952
  - module name, 952
- modulus operator, 84
- monitor, 2
- monospace font (constant-width characters), 34, 576
- moon missions, 362

motherboards, 4

MouseEvent class, 834, 924–928. *See also* ActionEvent class

moving averages, 398–400

moving files, 768

multidimensional arrays, 418

multiple driven classes, 314

multiple inheritance, 664–670, 672. *See also* inheritance, interfaces

multiple statements on a line, 968

multiplication

  its operator precedence, 86–87

  symbol, 35, 83

multiplicity values, 594

multithreading, 194

mutator methods, 253, 333–334

mutual recursion, 497

MVC. *See* Model-View-Controller

(MVC) design pattern

N

named constants, 180, 312–314
  basic features, 81
  for format strings, 417
  versus hardcoded constants, 82
  initializing, 296–299
  in interfaces, 665–666, 668, 672, 673
  in Math class, 177–181
  wrapper class, 183, 532, 533
naming rules
  for classes, 71, 74, 329
  for constructors, 294
  for methods, 73, 329
  overview, 73–74
  for variables, 35, 74–75, 329

  - for variables in overloaded methods, 289
  - for variables in separate blocks, 251
- NaN, 717
- NASA space program, 362–363
- nested looping structures, 48–51, 150–152, 412–413
- Netscape, 16
- network composition, 470
- new operator
  - in array instantiation, 389
  - basic purpose, 232–233
  - in constructor calls, 202, 301
- New Technology (NT), 360
- newBufferedreader method, 752–753
- newBufferedWriter method, 749–750, 751
- newline character, 93–94

next method, 102

nextBoolean method, 201–202

nextDouble method, 102, 105, 201

nextFloat method, 102

nextGaussian method, 201–203

nextInt method

- basic actions of, 85, 102, 105
- versus parseInt method, 697
- Random class, 175, 199–202
- source code heading, 176

nextLine method, 101–102, 104–105, 738–739

nextLong method, 102–104

nodes, 451

Node class, 778–801, 851

non-static method error messages, 311

non-void return types, 718

nonvolatile memory, 6

NoSuchElementException, 709

NoSuchFileException, 711, 713–714, 768

“not” operator, 131

Notepad, 19–21

null values

  as defaults, 237, 274

  terminating loops at, 656–657, 669, 671

  testing for, 641

NullPointerException, 657

NumberAxis class, 725, 727

NumberFormatException, 697–698, 700–702, 715

numeric data

  basic types, 78–80

  converting, 541

- operators for, 83

O

- Oak, 16
- Object class
  - equals method, 639–640, 642
  - overview, 638
  - toString method, 643
- ObjectInputStream class, 746–747
- ObjectOutputStream class, 747, 777–778
- objects 12, 26, 171
- object code
  - defined, 13–14
  - portability problem, 14
- object diagrams (UML), 982
- object file I/O
  - implementing, 743–747

  - reading a serializable object from a file, 745–747
  - storing Java objects in file, 743
  - updating version of written object, 747
  - writing a serializable object to a file, 744, 745

ObjectInputStream class, 746–747, 777, 779–780

object-oriented programming. *See also* inheritance, program design
  - argument passing, 249–250
  - array basics, 389
  - basics of, 171
  - calling object identification, 234–236
  - constructor overview, 293–299
  - driver classes, 230–234
  - instance variable default values and

persistence, 236–237
local variables overview, 244
method-call chaining, 287–289
modeling and implementing classes, 227–230
multiple driven classes, 314
multiple-method-program introduction, 213
object creation details, 273–274
overloaded constructors, 299–303
overloaded methods, 289–293
overview, 171, 223–227, 242–244
passing references as arguments, 284–286
simulation techniques in, 255–261
specialized methods in, 252–255
testing objects for equality, 279–284
tracing, 237–242

ObjectOutputStream class, 744–745, 747, 777–778

objects

- anonymous, 446–449
- arrays of, 418–424
- creating, 273–274
- defined, 96, 171, 224
- reference variables, relation to, 232–233
- representation in UML class diagram, 227
- retrieving all of an enumerated type, 563–564
- temporary, 276–279
- testing for equality, 279–284

octal numbers, 573–574

off-by-one errors, 44, 554

off-line storage, 8

offset continuous uniform distributions, 199, 205–206

one-line comments, 67

openConnection method, 741, 742, 752

opening braces, 70–71. *See also* braces

OpenOption interface, 749–751, 762

operands, 35, 83

operating systems, 19

operators

  cast, 86, 90–92

  common types in algorithms, 35

  comparison, 121

  compound assignment, 90–97

  increment and decrement, 88–90

  logical, 124–131

  for numeric data, 83–85

  precedence of, 35, 86–87, 126–127

Optional class, 107–108, 521

OptionalDouble class, 567, 571

“or” operator, 127, 129–130

Oracle, 17, 24

Oracle Database, 17

Oracle Java API website, 172–175, 359–360

output devices, 2–3

output formatting, 194–198

outsiders, documentation for, 334–338, 344–345

ovals, 216–217, 220–221

overflow errors, 532–533

overhead, 496, 517

overloaded constructors, 299–303, 606

overloaded methods, 289–293

overriding methods. *See* method overriding

P

packages
  - custom, 955, 959–960
  - defined, 25, 174, 949
  - hierarchical organization, 951–952, 955

Pane container, 851, 889–890. *See also* layout panes

parameters
  - defined, 215–216
  - versus local variables, 251, 342–343
  - in overloaded constructors, 299–303
  - in overloaded methods, 289–290

parent class, 602

parentheses
  - with calculated variables, 40
  - with cast operators, 92

  - for control-statement conditions, 121
  - as flag characters, 197
  - with logical operators, 126–127
  - in method calls, 99, 383–391
  - optional with return statements, 254
  - in switch constructs, 133–134
  - when to use, 35–36, 76, 391–393
- parseDouble method, 182
- parseFloat method, 182
- parseInt method
  - basic purpose, 182
  - potential errors with, 696–699, 701–702, 715
- parseLong method, 182
- partially filled arrays, 393
- pass-by-value, 251
- passed-in references, 284
- path. *See* directives

Path.createDirectory method, 768
path to directory, 22–23, 767
PathMatcher class, 771, 773
Paths class, 738, 748–750, 753, 761
Paths.get method, 708–711, 713
patterns, design, 346–347
percent symbol
  in compound assignment operator, 89–90
  as conversion specifier, 552, 574
  with format specifier, 195
  as modulus operator, 8
performance analysis, 514–517
persistence, 237, 247
PerspectiveCamera class, 677, 679–685
PhongMaterial class, 677, 679–683
photographs. *See* Image class.
PI constant, 179, 313

pixels, 204, 206–207, 218–221, 793–795

plain text editors, 19

plus sign

- as addition operator, 83, 86–87
- for concatenation, 72, 77–76, 93, 538
- as prefix in UML diagrams, 243
- as unary operator, 86–87

PointLight class, 682–684

polymorphism

- abstract methods and classes, 660–663
- with arrays, 654–660
- assignment between classes and, 653
- defined, 638
- with interfaces, 664, 666–667, 670
- overview, 638, 648–649

pop, stack, 460, 462, 463

pop method in Stack class, 463

port, 8

portability, 6, 14–15

position determination of string characters, 191

Pos class, 792–794, 798, 803–804

position method, 754, 755, 757, 759, 762, 765

postconditions, 334–335

postfix mode, 541–544

postVisitDirectory method, 745–747

pow method, 177

prebuilt methods. *See also* API library

  API library overview, 171–175

  Character class, 182

  Math class, 177–181

  printf, 194–198

  - random number generation, 199–203
  - String class, 176–183
  - StringBuilder and StringBuffer classes, 194
  - use in bottom-up design, 359–361
  - wrapper classes, 182–186
- precedence of operators
  - basic, 35, 86–87
  - in Boolean algebra, 158
  - with logical operators, 126–127
  - summary list, 945–946
- precision, 196–197, 533–536
- preconditions, 334–335
- prefix mode, 541–544
- preliminary class diagrams, 614–616
- preVisitDirectory method, 771–774
- primitive buffers with random access,

752–760

primitive data types. *See also* data types

  buffering, 727–734

  storing in ArrayList, 443–445

  wrapper classes, 182–184

  and wrapper types, converting between, 445

primitive variable examples, 95, 96

print statements

  avoiding in toString methods, 644

  for binary, hexadecimal, and octal equivalents, 574

  pseudocode, 33, 40

  using System.out, 101–104

  tracing, 52, 53

  unneeded, 253–253

PrinterJob class, 477–482

printf method, 194–199, 765, 768–769, 774

println method

  with charAt method, 98–99

  in nested loops, 151

  purpose, 71

PrintStream class, 567

PrintWriter class, 722–723

PrintWriter object creation, 737–738

PriorityQueue class, 465–467. *See also* Collection interface

privacy, 18

private access modifier

  for static variables, 304

  for helper methods, 339

  for instance variables, 228

  prohibited with abstract methods,

    663

private constructor in enumerated type, 561

procedural programming, 214

processActors method, 621

processor, 16, 19

program design. *See also algorithms,* object-oriented programming, tracing

    basic principles, 345–349

    bottom-up approach, 359–361

    case-based approach, 361

    iterative enhancement in, 361–363

    selecting class relationships, 613–618

    top-down approach, 350–359

programmer-defined exception classes, 672

programming languages, 12

programs, 2, 11, 13, 16. *See also* computer programs

programs, by class name except as noted. *See also* algorithms, programs with GUI

- adjustSalary method, 224–226
- ArrayCopy, 395. *Autonomous*
- Arrow, 946. *Autonomous*
- AssetAging interface, 671, 672
- AverageScore, 545. *Autonomous*
- Bear, 447
- BearStore, 448–449, 564–565. *Needs* Bear
- BinarySearch, 504
- BinarySearchDriver, 505. *Needs* BinarySearch
- binarySearch method, 407
- BonusCalculator, 76. *Autonomous*
- BridalRegistry, 141. *Autonomous*

BudgetReport, 198. *Autonomous*

BufferedReadFromFile, 753. *Autonomous*

BufferedWriteToFile, 750. *Autonomous*

ByteBufferAccess, 757. *Autonomous*

ByteBufferArrayAccess, 759. *Autonomous*

ByteOverflowDemo, 533. *Autonomous*

CalendarDemo, 369. *Autonomous*

Candide, 191. *Autonomous*

Car, 277, 596, 641

CarDriver, 278. *Needs* Car (first version)

Car2, 282, 645. *Autonomous second version*

Car2Driver, 288. *Needs* Car2 (first version)

Car3, 288

Car3Driver, 287. *Needs* Car3

Car4, 294

Car4Driver, 295. *Needs* Car4

Card (UML only), 615

Cat, 649

ChanneledFileAccess, 764–766. *Autonomous*

ChipotlesQueue, 459. *Autonomous*

Citizen, 474

City enumerated type, 562–563

CityTravel, 564. *Needs* City

CoinFlips, 402. *Autonomous*

Commission interface, 666

Commission2 interface, 666, 674

Commissioned, 667. *Needs* Employee2, Commission

Commissioned2, 667, 674. *Needs*

Employee3

Community, 471–472. *Needs* Citizen

CommunityDriver, 469. *Needs* Citizen, Community

ContactList, 388. *Autonomous*

ContactList2, 392. *Autonomous*

Counter, 646. *Autonomous*

Course, 404

CourseDriver, 405. *Needs* Course

CrabCritter2. *See* programs with GUI

CrabCritter2Runner. *See* programs with GUI

CreateNewFile, 706–708. *Autonomous*

Customer (UML only), 601, 627. *Needs* Person (second version)

DayTrader, 555. *Autonomous*

Dealership, 595

DealershipDriver, 598. *Needs* Dealership, Manager, SalesPerson

Dealership2, 612, 613. *Needs* Car (second version), Manager2, SalesPerson2

Dealership3 (UML only), 627, 628

Deck, 615, 617. *Needs* Card, GroupOfCards

DescartesDarwin, 193. *Autonomous*

DescartesDarwin2. *Autonomous*

DirectoryDescription, 769. *Autonomous*

Dog, 649

Dream, 67. *Autonomous*

DrivewayParking, 461–463. *Autonomous*

Employee, 296, 604, 658, 977. *Second version needs* Person (second version)

EmployeeDriver, 295. *Needs*

Employee (first version)

Employee2 (abstract), 297, 661

Employee2Driver, 297. *Needs* Employee2 (first version)

Employee3, 298, 675. *Needs* Commission2

Employee3Driver, 299. *Needs* Employee3 (first version)

Factorial, 147, 493. *Autonomous*

Factorial2, 495. *Autonomous*

FactorialIt, 496. *Autonomous*

FileVisitor, 773–774

FindFiles, 770. *Needs* FileVisitor

FindHypotenuse, 180. *Autonomous*

FlightTimes, 416–417

FlightTimesDriver, 415. *Needs* FlightTimes

FloorSpace, 144. *Autonomous*

Fraction, 300, 302

FractionDriver, 300. *Needs* Fraction

FreeFries, 128. *Autonomous*

FriendlyHello, 102. *Autonomous*

FullTime, 607. *Needs* Employee (second version)

FullTimeDriver, 609. *Needs* FullTime

Furniture. *Needs* AssetAging

Game, 615, 618. *Needs* Deck, Hand

GarageDoor, 155. *Autonomous*

getIntFromUser method, 700, 701.

Goodby, 97. *Autonomous*

GradeManagement, 559–560. *Autonomous*

GroupOfCards (UML only), 615. *Needs* Card

Growth, 257

GrowthDriver, 258. *Needs* Growth

Hamlet, 191. *Autonomous*

Hamlet2, 192–193. *Autonomous*

Hand (UML only), 615. *Needs* Card, GroupOfCards

Height, 291

HeightDriver, 292. *Needs* Height

Hello, 20. *Autonomous*

HorseFly, 97. *Autonomous*

Hourly, 660. *Needs* Employee

Hourly2, 662. *Needs* Employee2

Hourly3, 662, 674. *Needs* Employee3

HTMLGenerator, 738–739. *Autonomous*

Human, 313

HungerGames, 442. *Autonomous*

IdentifierChecker, 187. *Autonomous*

insertionSort method, 515

LinePlot, 696, 698. *Autonomous*

ListExecutionTimes, 454–455. *Autonomous*

Lottery, 184–185. *Autonomous*

LuckyNumber, 694. *Autonomous*

MacBeth, 192. *Autonomous*

Manager, 597. *Needs* Person (second version)

Manager2, 613. *Needs* Person (second version)

MergeSort, 507–509. *Autonomous*

MethodPromotion, 540. *Autonomous*

module-info, 956, 958, 960, 962

Mouse, 229, 239

MouseDriver, 231. *Needs* Mouse

MouseDriver2, 238. *Needs* Mouse

Mouse2, 246

Mouse2Driver, 245. *Needs* Mouse2

Mouse3, 250

Mouse3Driver, 250. *Needs* Mouse3

MouseShortcut, 366

MovingAverage, 399. *Autonomous*

NestedLoopRectangle, 151. *Autonomous*

NumberList, 716

NumberListDriver, 715. *Needs* NumberList

Payroll, 656. *Needs* Employee, Hourly, Salaried

Payroll2, 662. *Needs* Employee2, Hourly2, Salaried2

Payroll3, 669. *Needs* Commission, Commissioned, Employee2, Hourly2, Salaried2, SalariedAndCommissioned

Payroll4, 671. *Needs* Commission, Commissioned, Employee2, SalariedAndCommissioned

Payroll5, 674. *Needs*

Commissioned2, Salaried3, Hourly3, SalariedAndCommissioned2

PennyJar, 308

PennyJarDriver, 309. *Needs* PennyJar

Person, 286, 603

PersonDriver, 285. *Needs* Person (first version)

Person2, 572

Person2Driver, 573. *Needs* Person2

Pets, 651. *Needs* Cat, Dog

Pets2, 652. *Needs* Cat, Dog

phoneList array, 385–389

PrintCharFromAscii, 542. *Autonomous*

PrintInitials, 104. *Autonomous*

PrintInitials2, 106. *Autonomous*

PrintPO, 103. *Autonomous*

PrintReverseMessageIterative, 498. *Autonomous*

PrintReverseMessageRecursive, 498–499. *Autonomous*

PrintUtilities, 367

RandomTest, 202. *Autonomous*

ReadFromFile, 710. *Autonomous*

ReadFromFile2, 713. *Autonomous*

ReadObject, 746. *Needs* TestObject

RollDice, 214–215. *Autonomous*

Salaried, 659. *Needs* Employee

Salaried2, 662. *Needs* Employee2

Salaried3, 662, 674, 676. *Needs* Employee3

Salaried_jd, 977. *Needs* Employee

SalariedAndCommissioned, 668. *Needs* Commission, Salaried2

SalariedAndCommissioned2, 668, 674, 676. *Needs* Salaried3

Sale, 628. *Needs* Car (second version), Customer (second version), SalesPerson2

SalesClerk, 423

SalesClerks, 421–422. *Needs* SalesClerk

SalesClerksDriver, 424. *Needs* SalesClerks

SalesPerson, 597. *Needs* Person (second version)

SalesPerson2, 613. *Needs* Person (second version)

Sayings, 72. *Autonomous*

SentenceTester, 125. *Autonomous*

Shirt, 340–341

ShirtAssert, 349

ShirtDriver, 339. *Needs* Shirt

ShootingPercentage, 551. *Autonomous*

Sort, 411

SortDriver, 412. *Needs* Sort

SortingCourseDriver, 407. *Needs* Utilities

Square, 353–354, 357–358

SquareDriver, 352. *Needs* Square

StockAverage, 444. *Autonomous*

StockAverage2, 568. *Autonomous*

StringMethodDemo, 190. *Autonomous*

Student, 327–328, 972

StudentDriver, 326, 974. *Needs* Student

Student_jd, 327–328, 337, 977.

StudentList, 703, 705

StudentListDriver, 704. *Needs* StudentList

StudentList2, 719

StudentList2Driver, 721. *Needs* StudentList2

TemperatureConverter, 83. *Autonomous*

Test, 98. *Anonymous*

TestExpressions, 85. *Autonomous*

TestObject, 744

TestOperators, 91. *Autonomous*

Time, 364. *Autonomous*

Towers (of Hanoi), 512. *Autonomous*

TruthTable, 160. *Autonomous*

Utilities (for binary search), 407

WebPageReader, 742. *Autonomous*

WriteObject, 745. *Needs* TestObject

WritetoFile, 723. *Autonomous*

ZipCodeExpression, 137. *Autonomous*

ZIPCodeStatement, 136. *Autonomous*

ZIPCodeTraditionalSwitch, 139. *Autonomous*

programs with GUI, by class name except as noted. *See also* algorithms, programs

AdjustableReader, 920–921, 922. *Autonomous*

AfricanCountries, 870. *Autonomous*

AfricanCountries2, 871. *Autonomous*

CardData, 776

ColorChooser, 840–841. *Autonomous*

Combo, 910–913, *Autonomous*

CrabCritter2, 623–625

CrabCritter2Runner, 626. *Needs* CrabCritter2, and Bug, Flower, Rock (from GridWorld)

CRC_Card, 372

CRC_Card2, 372, 775

CRCDriver, 374. *Needs* CRC_Card

CRCDriver2, 478–481. *Needs* CRC_Card

CRCDriver3, 478–481, 778–779. *Needs* CardData, CRC_Card2

Descartes, 814–815. *Autonomous*

Descartes2, 814, 820. *Autonomous*

Descartes3, 822, 824. *Autonomous*

DialogDemo, 108–111. *Autonomous*

Dorm Life. *See* AdjustableReader

DragSmiley, 924–925, 927, 928. *Needs* two image files

FactorialButton, 829, 830–831. *Autonomous*

FlowPaneDemo, 873, 874. *Autonomous*

GraphicsDemoC, 205. *Autonomous*

GraphicsDemoJ, 218–221. *Autonomous*

Greeting, 803–804, 808. *Autonomous*

GreetingGrid, 855–856. *Autonomous*

GreetingLambda, 803–804, 808. *Autonomous*

GreetingPropertyBinding, 810–811. *Autonomous*

handle method, 831, 833

Installation, 907–908, 909. *Autonomous*

JobApplication, 915–917, 917. *Autonomous*

LabelHello, 27. *Autonomous*

License, 901, 902. *Autonomous*

LinePlotGUI, 725–727. *Autonomous*

LunarEclipse, 930, 931–932. *Autonomous*

LunarEclipseAnimate, 930, 934–936. *Autonomous*

MathCalculator, 885–888, 889. *Autonomous*

Recital, 861–863, 865. *Needs* an image file

RotatedCylinder, 679–680. *Autonomous*

SecondaryStage, 795–796

Signpost, 682–684, 679–680. *Autonomous*

SimpleWindow, 792. *Autonomous*

start method for DragSmileyLambda 927

TextFlowDemo, 874, 877. *Autonomous*

TicTacToe, 880–881, 882. *Autonomous*

TilePaneDemo, 874, 875. *Autonomous*

TitleHello, 25. *Autonomous*

TreeController, 522. *Needs* TreeModel, TreeView

TreeModel, 519

- TreeView, 520
- UnicodeDisplay, 577. *Autonomous*

Project Apollo, 363

Project Gemini, 363

Project Mercury, 363

prologues
- coding conventions for, 326
- overview, 68, 326

promotion of operands, 84, 539–540

prompts, 22, 36

property binding, 789, 809–812

property values, 816–818, 825. *See also* CSS properties

protected access modifier, 673–677

prototyping, 362

pseudocode. *See also* algorithms
- converting to source code, 12
- defined, 11, 33

  - indentation in, 39
  - proceeding without, 13
  - versus programming code, 66
  - varieties of, 55–56
- public access modifier
  - defined, 69, 70
  - for static constants, 313
  - for instance variables, 228, 744
- public methods, 350–353, 357, 359, 464
- push, stacks, 460–463
- putInt method, 755–757

Q

- queries
  - with do versus while loops, 143–144, 149
  - in nested loops, 49
  - terminating loops with, 46–47

Queue interface, 459–460, 467

queues, 436, 457–460

quotation marks, 71, 94, 95

R

RadialGradient class, 204–206, 851, 928–933

RadioButton control, 851, 905–906

radio button, 897

random access, primitive buffers with, 752–760

Random Access Memory (RAM), 6

Random class, 171, 175, 179, 201–203, 442, 454–455, 471

random method. *See* Math.random method

random numbers, 199–203, 215

read operations

  BufferedReader, 752

  - FileChannel methods, 761–763
  - reading a serializable object from a file, 745–747
  - website reading, 741–742, 752

- readability, 68, 70–71, 74
- read-only memory, 8
- ready method in BufferedReader class, 752–753
- realizable types, 600
- record, 763–764
- Rectangle class, 204–206, 851
- rectangles
  - drawing, 216–221
  - in flowcharts, 37–38
- Rectangle2D class, 218, 221
- recurrence relations, 492, 494, 500–502

recursion, 490, 1009

  types of, 496–497

recursive algorithms, expressed as functions, 500–502

recursion example. *See* recursion example

recursive methods

  evaluation of, 500–503

  factorial method, 493–495

  versus iterative solutions, 496–500

  overview, 490–491

  writing, 491–492

redundancy, avoiding

  with generic catch, 714

  with helper methods, 341

  with looping structures, 43

  with inheritance, 602

reference types, 96, 652–653

reference variables
  versus anonymous objects, 446
  ArrayList initialization, 437
  assigning values to, 233, 273–279
  charAt and println methods, 98
  copying, 275–276
  declaring, 233
  default value, 236–237, 264
  instantiating, 233, 264
  null, 639, 641, 656–657
  versus objects, 232–233
  omitting dot prefixes with, 342
  overview, 96, 232–233
  passing as arguments, 284–286
  polymorphism and, 638
  two with one method call, 281–282
relational database tables, 763–767

relational operators, 154–156

relative get and put methods, 755–756

relative paths, 767–768

reliability, 18

repetitive tasks, 138

replaceAll method in String class, 189, 192

replaceFirst method in String class, 189, 192

replaceSelection method, 900

replacement of text, 192

requirements analysis, 10, 11

requires. *See* directives

reserved words, 69, 947–950

reserved type name. *See* local variable type inference

reset method, 747, 754

resolution (screen), 793

retrieving all objects in an enumerated type, 563–564

return reserve word, 214–215, 219–221

return statement, 247–249, 254

return this statement, 287–288

return types. *See also* data types

  matching method headings, 247–249

  omitting from constructor headings, 294

  for overriding methods, 608

return values

  boolean, 237, 253–255, 281

  defined, 176

  return statement overview, 247–249

reverseList method in List interface, 453

rewind method, 754–757, 759, 764

RGB value, 834–836, 840–841. *See also* colors

Rich Internet Application (RIA), 790

right-hand rule, 679

robustness of algorithms, 48

ROM, 8

Rotate class, 677, 679–684

random method. *See* Math.random method

RoundingMode class, 535

round-off errors, 534

round method. *See* Math.round method

row variable, 75

rows

  in a table, 763–764

  in GridPane container, 854, 856

  in layout panes, 852

run commands, 13

runtime errors
    analyzing, 714–718
    defined, 99, 131, 692, 676, 704
    non-integer error, 715
RuntimeException errors, 701–702, 708
Russell 3000 Index, 443

S

sample sessions, 36
Save As dialog box, 20–21
saving files, 20–21, 73
scalability, 418
Scanner class, 100–105, 171–172, 175–176
Scanner constructor calls, 101–106, 709–710, 738
Scanner methods, 102, 738
Scanner object, 105–108

Scene class, 26, 204–205, 791–795, 851

scene graph, 521, 801, 803, 812, 818–819

scene graph inheritance, 818–821

scene graph root, 801, 819, 821

scientific notation, 535

scope, 244, 306

screen, 3

screen resolution, 793

screen shots, 11

scrollable containers, 919, 920

ScrollPane class, 851, 919–919

searching an array, 403–408

secondary storage, 4

section delimiting, 963

SecurityException class, 708

seed, 203
selection sort, 408–410, 517
selector, 823
  class selector, 838
  type selector, 839
self-documenting code, 74, 964
self-similarity, 518
semantics, 122–123
semicolons
  after do loops, 135
  after switch expressions, 133–134
  for empty statements, 552–553
  required for Java statements, 13, 71, 74
sentinel values
  boolean variables as, 153
  in nested loops, 48–51
  “q” as, 667

  terminating loops with, 45, 47–48

separation of concerns, 346

sequential search, 403–406, 517

sequential structures, 37, 66

Sequential-execution programs, 66, 120

Serializable interface, 743–744, 776

serialization, 743–746

servers, 338

servlets, 16

Set interface, 467. *See also* Collection interface

setAlertType method, 111

setEditable method, 899–901

setOnAction method, 801, 804, 805, 807–809

set methods

  ArrayList, 439–441, 455–456, 517

  - defined, 253
  - LinkedList, 455–456, 517

sets, 464

setScene method, 27, 791–795

setStyle method, 812–813

setTitle method, 107–110, 792–795

Shape3D subclasses. *See* Box, Cylinder, Sphere

shared values, static variables for, 304

shifting array element values, 397–398

short type, 532

short-circuit evaluation, 551–552, 641

ShortBuffer class, 752, 758

short-form tracing, 52–53. *See also* tracing

show method, 25–27, 792–793

showAndWait method, 107–111

signature, 289

significant digits, 80

SimpleFileVisitor, 771, 773

simulation, 255–261, 401–403

sin method, 179, 181

single quotes, 93–95, 135

16-bit characters, 539. *See also* hexits

16-bit instructions, 14

size method, 761, 762

SKIP_SUBTREE option, 773

slashes. *See also* division

  in compound assignment operator, 89–90

  as division operator, 13, 35

  in path directories, 767, 770

  to set off comments, 67

sleep method, 553

Slider class, 851, 928–933

software development tools, 55, 242

software engineering, 325

solid-state drives, 7–8

Sort class, 401, 402

sort method, 403

sorting arrays, 408–412

sorting characters, 536–537

source code, 12–13

  compiling to object code, 13–14

source code for authors' programs. *See* programs

source code for authors' GUI programs. *See* programs with GUI

space program example, 362–363

spaces, 34–35, 332–333, 968

spaghetti code, 37

specifications, 350, 360, 362

Sphere class, 677, 682–683

spherical shape, 684

SplitPane container, 373, 851

splitting string literals, 72

spreadsheet table, 763

square brackets

- with ArrayList, 443
- in array declarations, 387
- in arrays with two or more dimensions, 412–413, 418
- in for-each loop, 425–426
- purpose, 70, 196

stack overflow, 495

StackPane container, 218–220, 850–852

stacks, 436, 460–464

Stage class

- primary stage, 25–27, 107–108, 204–205, 794–795, 851
- secondary stage, 795–796, 913, 915,

917

StageStyle class, 25, 796

standard coding conventions, 69

Standard Edition applications, 17

standard windows. *See* windows (GUI)

StandardOpenOption, 749–751. *See also* enum, OpenOption

start method, 25–27, 791–793. *See also* Application class

start tags, 739–741

state variables, 343, 373

statements

- braces with, 329–331

- defined, 13
- flow of control, 37–38
- line breaks in, 330
- multiple statements in one line, 968

- pseudocode, 33–34

states
- of objects, defined, 224
- representing with instance variables, 350
- tracking with Boolean variables, 153

static allocation, 304

static binding, 660

static constants, 180, 313–314

static fields, 225, 303–306

static methods
- defined, 171
- identifying, 647
- in Character class, 186
- in interfaces, 665, 670–672
- searches with, 406
- when to use, 311–312

static modifier, 177, 180, 188, 202

- basic purpose, 70
- for static variables, 304
- in UML class diagrams, 243
- optional for interfaces, 665
- with Math methods, 177, 178

static named constants, 180, 313–314

stdIn variable, 101

stepping over method calls, 233

stepwise refinement, 350

step-with-midpoint algorithm, 259–261

Stop class, 204–205, 928, 931–935

stopping condition, 491

storage capacity comparison, 8

storage devices, 6–8

Stream classes and constructs

- DoubleStream, 567, 569–571
- FileInputStream, 746, 777, 779–780

  - FileOutputStream, 744–745, 777–779
  - InputStream, 741–742, 752
  - InputStreamReader, 752
  - IntegerStream, 569
  - ObjectOutputStream, 746–747, 777–779
  - Steam.Builder, 569
  - stream operations, 569–571
  - stream pipeline, 567
  - stream programming, 567–573
- String class, 96–100, 188–194
- String[] arguments, 26, 70
- String concatenation, 75–76, 93, 97
- String's length method, 391–393
- String literals, 34, 71–72
- String methods, 97–100, 188–193
- string pooling, 643

string reversing alternatives, 497–499

String variable, 129, 135, 142

String class, 70, 74, 96

String[] arguments, 70

strings

- Buffer methods and, 749, 758–760
- versus char type, 93
- comparing, 129, 280–283
- concatenating, 72, 93, 97, 141–142, 548
- converting from byte arrays, 758–760
- converting from primitive types, 182–183, 697
- declaration syntax, 74
- defined, 34, 70
- escape sequences in, 93–95
- Java features for manipulating, 96–

    99
  - parsing, 182–183, 185, 831–832, 838
  - printing in reverse order, 497–500
- strongly typed languages, 77, 538
- structured programming, 37
- stubs, 350, 353–354
- style conventions, 325–334
- style sheet, 821–825
- styling, 812–818
- subclasses, 601–602
- subordinate statements, 38–39, 121–122
- subscripting, 386
- substring method in String class, 189
- substring retrieval, 191
- subtraction, 13, 35, 83, 87
- subtree, 675, 676, 677

Sun Microsystems, 16, 17
superclasses
  basic features, 601–602
  constructors in, 605–606
  implementing, 603–604
  initial diagramming, 614–616
  method overriding and, 606–608
super constructor call, 605–606
swimlanes, 981–982
Swing, 789–790. *See also* AWT
switch constructs, 132–138
  block, 133
  expressions, 133–134, 137
  header, 133
  overview, 132
  statements, 132–133, 136
  traditional statements, 137–138

  - versus “if, else-if”, 138
- syntax
  - array and ArrayList, 387, 389, 393, 412, 418, 425, 437, 438, 449
  - cast operators, 540–542
  - charAt and println methods, 98
  - comment, 67–68, 332
  - conditional operator expressions, 546
  - declaration statements, 74
  - defined, 12
  - equals method, 99–100, 123, 130, 639
  - for-each loop, 425–426
  - format specifier, 195–196
  - importance to programming, 12

  - instance constants, 297

  - instance method calls, 233–234
  - Java versus pseudocode, 66
  - looping structures, 138, 142, 144, 149
  - overloaded constructors, 301
  - prebuilt method calls, 170–171, 176
  - serialization declaration, 743–744
  - static constants, 313
  - static methods, 308
  - static variables, 304
  - string–primitive conversions, 182–183
  - try and catch blocks, 693–695
- syntax errors, 23
- system reliability, 18
- System class, 176
- System.arraycopy method, 396, 398, 400

System.nanoTime method, 453–455

System.out.print statement, 98, 709

System.out.println statement, 71–72, 101

T

tab character, 93–94

table formats, 763

tablet computers, 7

tags in HTML, 737, 739–741

tail of list, 452

tail recursion, 497, 499

tan method, 179, 181

temporary objects, 276, 279

temporary print statements, 252–253

temporary variables, 285, 286

terabyte, 9

terminate abnormally, 99

termination of loops
    basic principles, 45–46
    with boolean variables, 146
    common techniques, 45–48
    do loops, 143
    length property for, 413–414, 418
    with return statements, 249
ternary operators, 546
testing. *See also* tracing
    defined, 10
    freezing random numbers for, 203
    local main methods for, 365
    overview, 347
    standard input data for, 363
text, adding to graphics, 220–221, 851
TextArea control, 372–373, 576–577, 796, 817, 851
TextField control, 372–373, 799–801,

851

TextFlow container, 851–852, 876–878

TextInputDialog class, 107–109, 480, 851

text boxes, 107, 788–789, 799–801. *See also* TextField, TextInputDialog

Text class, 218, 220–221, 520–521

text editor, 19

text I/O

  buffered text files, 749–752

  HTMLGenerator example, 737–741

text replacement, 192

TextField, 799–801

TextInputDialog, 107–109

this constructor calls, 301

this reference

  as calling object identifier, 234–236, 292

  - as instance variable identifier, 230, 234, 235–236
  - omitting, 365–366

threads, 194

three-dimensional graphics, 677–685

throwing an exception, 695

throws clause, 718–720

throws Exception, 105–106

thumb, 928

thumb drives, 8

TilePane container, 851–852, 872–876

Timeline class, 934–936

time complexity analysis, 514

times and dates, 368

tinted pane, covering image with, 203–206

title bars, 24–25, 107

ToggleGroup class, 905–908, 915–916

tokens, 102–103

toLowerCase method, 186, 189, 192

top of stack, 436, 460

top-down design, 350–359

toString methods, 182–183, 469–470, 473–474, 574, 643–648

toUpperCase method, 186, 189, 192

Towers of Hanoi, 510–514, 517

tracing. *See also* testing

  with constructors, 302–303

  object-oriented programs, 237–242

  of operations, 90

  in pseudocode, 33, 51–55

  recursive factorial method, 494–495

  setup for, 76–77, 146–147

Translate class, 679–684

TreeSet class, 465–466, 468, 473–474, 517 trigonometric math methods, 179, 181

trim method, 189, 192–193

true reserved word, 120–121

TRUNCATE_EXISTING option, 749–751

truth tables, 157–158

try blocks

  in buffered I/O, 750–752

  in file channel I/O, 766

  deleting files, 768

  details in implementing, 699–700

  for input validation, 697–698

  not for initialization, 700

  overview, 693–695

try-catch structures

  for checked exceptions, 705–707

  with generic catch block, 709

  - for input validation, 697–698
  - moving to calling method, 718–720
  - with multiple catch blocks, 712–714
  - overview, 693–695
  - for unchecked exceptions, 702–705
- try-with-resources header, 720–722, 737, 746
- two-dimensional arrays, 412–418, 517
- type casting, 90–92, 538–539, 540–541
- type inferencing, 437–438
- types. *See* data types

U

- UML class diagrams, 980–985
  - association in, 594, 627
  - for array of objects, 420
  - basic features, 227–228, 242–244
  - composition and aggregation in,

594, 611, 612, 615, 627
    inheritance in, 600–602, 655, 674
    use in top-down design, 352, 356
    Card Game program, 615
    Dealership program, 594, 611, 627
    Mouse program, 227, 243
    Payroll program, 655, 674
    PennyJar program, 307
    SalesClerks program, 420
    Square program, 352, 356
unary operators, 86–87
unboxing, 444–445
unchecked exceptions, 702–705, 714
underlining in UML diagrams, 243
Unicode characters
    versus ASCII values, 537, 574–575
    character sets and, 748

  - need for, 537
  - overview, 574–575
  - website, 575*n*, 576*n*, 578

Unified Modeling Language (UML), 980. *See* UML class diagrams

unintended empty statements, 553–554

universal constants, 673

universal serial bus, 7–8

UNIX, 19

unserialization, 743, 745–746

UnsupportedOperationsException errors, 708

update component of for loop, 145–146

uppercase letters. *See* capital letters

urban legend, 10

URL class, 741–742, 752

URLConnection class, 741–742, 752

US-ASCII, 748, 751

USB flash drives, 8

user, 33, 36, 344–345

User Interface Design (UID), 897–898

user queries. *See* queries

user-friendliness of Boolean variables, 154

UTF-8 character set, 741, 748

UTF-16 character set, 748

utility methods, 312

V

valueOf method, 648, 750–751

varargs, 751

variables

  arrays as, 387

  assignment syntax, 75–78

  - Boolean, 152–156
  - converting data type, 90–91
  - declaration syntax, 74–75, 452–453
  - defined, 34
  - garbage values, 90
  - index, 146–148, 151–152, 244
  - initialization syntax, 77–78
  - local, 227, 240, 242, 244
  - naming, 34–35, 251, 329
  - numeric data types for, 78–80
  - primitive versus reference, 95–96, 232–233
  - random number generation, 199–202
  - scope of, 148
  - static versus instance types, 226
  - temporary, 285, 286

VBox container, 372–373, 850–852,

858–866

view, 518, 520–521. *See also* Model-View-Controller (MVC)

visitFile method, 771, 774

void modifier, 70, 235, 236, 238–239

volatile memory, 6

W

walkFileTree method, 770–771

website reading, 741–742, 752–753

websites

  Computer Ethics Institute, 18

  Computer Professionals for Social Responsibility, 18

  GridWorld software, 579*n*

  Java Code Conventions, 325*n*

  Java documentation, 97, 171–172, 359

  JDK installation instructions, 23

Unicode, 575*n*, 576*n*, 578

Webopedia, 2

while loops

assignment statements in, 546

for input validation, 156–157

in pseudocode, 44–48

in try blocks, 711

overview, 120, 140

when to use, 149–150

with ArrayDeque, 459

while statements, 120, 138–142

white space

above comments, 332

defined, 102

nextLine method and, 102

removing or inserting, 192–193

width and height parameters, 219

wildcards, 176, 368

windows (GUI). *See* Graphical User Interfaces (GUIs)

Windows operating system, 360

Page 1014

wireframe, 883

word processors, 19

World Wide Web, 16. *See also* websites

wrapper classes

  with ArrayList, 443–445

  basic features, 182–186

  Character class, 186

  for floating-point numbers, 182, 533–536

  for integers, 182, 532–533, 535–536

  toString methods, 646–647

wrapper objects, 443, 445

write operations

  - BufferedWriter, 749–751
  - FileChannel methods, 760–763
  - updating version of written object, 747
  - writing a serializable object to a file, 744–745

X

- x and y coordinates of a circle, 206

Y

- yield, 134. *See also* switch constructs / expressions

Z

- zero
  - division by, 48, 301, 717
  - as flag character, 197
  - starting counts at, 46, 386–387, 393
- zero-parameter constructors, 295–296, 604, 606, 729

McGraw
Hill
Education

5011111